MethodsX 12 (2024) 102648

Contents lists available at ScienceDirect

MethodsX

journal homepage: www.elsevier.com/locate/methodsx

A step-by-step method to quantify coloration with digital photography ☆

Carolyne Houle [a], Audrey Turcotte [b,*], James E. Paterson [c], Gabriel Blouin-Demers [b], Dany Garant [a]

[a] *Département de biologie, Université de Sherbrooke, 2500 boulevard de l'Université, Sherbrooke, Québec, J1K 2R1, Canada*
[b] *Département de biologie, Université d'Ottawa, 30 Marie Curie, Ottawa, Ontario, K1N 6N5, Canada*
[c] *Institute for Wetland and Waterfowl Research, Ducks Unlimited Canada, 1 Mallard Bay, Stonewall, Manitoba, R0C 2Z0, Canada*

ARTICLE INFO

Method name:
Quantification of coloration using digital photography

Keywords:
Color binarization
Reflectance
Linearization
Color segmentation
Colorimetric value
Color measurement
Image analysis

ABSTRACT

Coloration is often used in biological studies, for example when studying social signaling or antipredator defense. Yet, few detailed and standardized methods are available to measure coloration using digital photography. Here we provide a step-by-step guide to help researchers quantify coloration from digital images. We first identify the do's and don'ts of taking pictures for coloration analysis. We then describe how to i) extract reflectance values with the software ImageJ; ii) fit and apply linearization equations to reflectance values; iii) scale and select the areas of interest in ImageJ; iv) standardize pictures; and v) binarize and measure the proportion of different colors in an area of interest. We apply our methodological protocol to digital pictures of painted turtles (*Chrysemys picta*), but the approach could be easily adapted to any species. More specifically, we wished to calculate the proportion of red and yellow on the neck and head of turtles. With this protocol, our main aims are to make coloration analyses with digital photography:

- More accessible to researchers without a background in photography.
- More consistent between studies.

Specifications Table

Subject area	Agricultural and Biological Sciences
More specific subject area	Coloration analyses in biology
Name of your method	Quantification of coloration using digital photography
Name and reference of original method	M. Stevens, C.A. Párraga, I.C. Cuthill, J.C. Partridge, T.S. Troscianko, Using digital photography to study animal coloration, Biol. J. Linn. Soc. 90 (2007) 211–237. J.E. Paterson, G. Blouin-Demers, Distinguishing discrete polymorphism from continuous variation in throat colour of tree lizards, *Urosaurus ornatus*, Biol. J. Linn. Soc. 121 (2017) 72–81. https://doi.org/10.1093/biolinnean/blw024.

☆ A. Turcotte, Effects of human disturbance and human-made barriers on the behaviour, physiology, and genetic structure of painted turtle populations, Doctoral theses, University of Ottawa, Ottawa (2023) http://dx.doi.org/10.20381/ruor-29834.

* Corresponding author.
E-mail address: aturc096@uottawa.ca (A. Turcotte).

https://doi.org/10.1016/j.mex.2024.102648
Received 11 October 2023; Accepted 5 March 2024
Available online 8 March 2024

	L.C. Teasdale, M. Stevens, D. Stuart-Fox, Discrete colour polymorphism in the tawny dragon lizard (*Ctenophorus decresii*) and differences in signal conspicuousness among morphs, J. Evol. Biol. 26 (2013) 1035–1046. https://doi.org/10.1111/jeb.12115. J. Troscianko, M. Stevens, Image calibration and analysis toolbox - a free software suite for objectively measuring reflectance, colour and pattern, Methods Ecol. Evol. 6 (2015) 1320–1331. https://doi.org/10.1111/2041-210X.12439. A. Turcotte, Effects of human disturbance and human-made barriers on the behaviour, physiology, and genetic structure of painted turtle populations, Doctoral theses, University of Ottawa, Ottawa (2023) http://dx.doi.org/10.20381/ruor-29834.
Resource availability	If applicable, include links to the resources necessary to reproduce your method (e.g., equipment, data, software, hardware, reagents). M.D. Abràmoff, P.J. Magalhães, S.J. Ram, Image Processing with ImageJ, Biophotonics Int. 11 (2004) 36–42. https://imagej.nih.gov/ij/download.html D. Coffin, IJ: Plugins: ij-dcraw – Reader for digital camera raw image. (2015) https://ij-plugins.sourceforge.net/plugins/dcraw/ R Core Team, R: A language and environment for statistical computing, R Foundation for Statistical Computing. (2022) https://www.R-project.org/ Posit Team, RStudio: Integrated Development Environment for R. Posit Software, PBC, Boston, MA. (2022) http://www.posit.co/. All files (i.e., Data, R codes, pictures) necessary to replicate all the steps of this methods are available in supplemental material.

About

Coloration is often used in biological studies, for example when studying social signaling or antipredator defense (see Cuthill et al. [1] for a review). Yet, few detailed standardized methods are available to measure coloration using digital photography. Here we aim to make coloration analyses: i) more accessible to researchers without a background in photography and ii) more consistent between studies. In this tutorial, you will learn 1) the do's and don'ts of taking pictures for coloration analysis, 2) to extract reflectance values with ImageJ, 3) to create linearization equations, 4) to linearize your pictures with the equations created, 5) to scale and select the areas of interest in ImageJ, 6) to rapidly classify your pictures, 7) to standardize your pictures, and 8) to binarize the colors of the areas of interest. Make sure you thoroughly read the whole protocol before beginning your analyses. Important notes are included throughout the different sections.

You should first install the following software/packages:

ImageJ: https://imagej.nih.gov/ij/download.html [2].

DCRaw Reader (ImageJ package to open various RAW file types): http://ij-plugins.sourceforge.net/plugins/dcraw/ and https://github.com/ij-plugins/ijp-dcraw [3].

R software: https://cran.r-project.org/ [4].

RStudio: https://posit.co/download/rstudio-desktop/ [5].

The tutorial has been optimized for use in Windows 10. We have not verified compatibility with any other operating system or version, but because the software and packages we used are available for multiple platforms, it should work on all major platforms such as Mac OS and Linux.

Basic knowledge of coloration and digital photography is needed to understand this tutorial. We suggest reading the following papers:

- M. Stevens, C.A. Párraga, I.C. Cuthill, J.C. Partridge, T.S. Troscianko, Using digital photography to study animal coloration, Biol. J. Linn. Soc. 90 (2007) 211–237 [6].
- J.E. Paterson, G. Blouin-Demers, Distinguishing discrete polymorphism from continuous variation in throat colour of tree lizards, *Urosaurus ornatus*, Biol. J. Linn. Soc. 121 (2017) 72–81. https://doi.org/10.1093/biolinnean/blw024 [7].
- L.C. Teasdale, M. Stevens, D. Stuart-Fox, Discrete colour polymorphism in the tawny dragon lizard (*Ctenophorus decresii*) and differences in signal conspicuousness among morphs, J. Evol. Biol. 26 (2013) 1035–1046. https://doi.org/10.1111/jeb.12115 [8].

All files necessary to run our scripts (e.g., Excel file, pictures, R codes) are available in supplementary material.

In the field: steps to follow

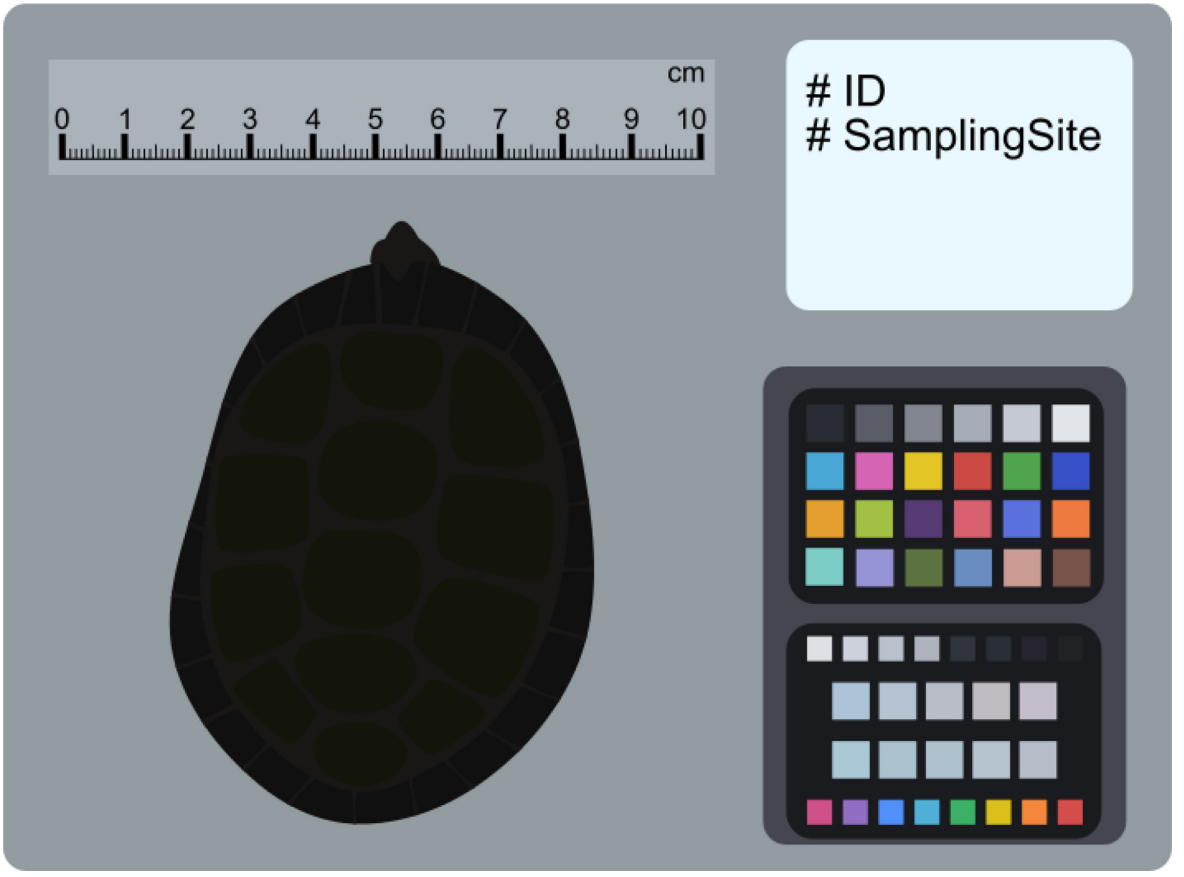
Here are some basic guidelines to consider before taking pictures:

Tip #1. All pictures should include a color chart (e.g., X-Rite ColorChecker Passport) and a ruler (most color charts already include a ruler). Color charts have standard color sections with known reflectance to compare to the pixel reflectance values in a picture of the same color.

Tip #2. Pictures should be stored as .RAW or an equivalent format (e.g., .NEF); avoid compressed files. Compressed file formats, including JPEG, lower the range and variation in color an image can store.

Tip #3. All pictures should be taken with identical camera settings. Avoid automatic settings. Importantly, make sure all your pictures are taken with the same ISO value.

Tip #4. It is more convenient if all picture elements have the same configuration: the object (e.g., animal) and the ruler are approximately at the same place in all pictures, and they are all taken either in portrait or in landscape format.



Tip #5. Take slightly underexposed pictures to avoid pixel saturation (see Stevens et al. [6]). Avoid direct sunlight to avoid pixel saturation; take your pictures in the shade or use a cover (e.g., umbrella). Pixel saturation is when the pixel color values reach the maximum recordable value.

Tip #6. Make sure you have access to the expected reflectance values (sRGB: Red (R), Green (G), Blue (B)) of the chart colors used. The expected grey reflectance values are necessary to create the linearization equations and, thus, essential to linearize your pictures correctly.

In our case, pictures were stored as .NEF and they all included an X-Rite ColorChecker Passport for which the grey reflectance values in sRGB are shown in the chart below (squares 19 to 24, scale between 0 and 255: 8-bit format; see X-Rite Incorporated [9]).

No.	Color name	Expected reflectance value (sRGB) R	G	B
1	dark skin	115	82	68
2	light skin	194	150	130
3	blue sky	98	122	157
4	foliage	87	108	67
5	blue flower	133	128	177
6	bluish green	103	189	170
7	orange	214	126	44
8	purplish blue	80	91	166
9	moderate red	193	90	99
10	purple	94	60	108
11	yellow green	157	188	64
12	orange yellow	224	163	46
13	blue	56	61	150
14	green	70	148	73
15	red	175	54	60
16	yellow	231	199	31
17	magenta	187	86	149
18	cyan	8	133	161
19	white	243	243	242
20	neutral 8	200	200	200
21	neutral 6.5	160	160	160
22	neutral 5	122	122	121
23	neutral 3.5	85	85	85
24	black	52	52	52

Extracting reflectance values with ImageJ

Preparing a spreadsheet file

Camera sensors record data on three color channels (RGB: Red (R), Green (G), Blue (B)) that can vary in reflectance value format (0–1, or another scale like 8-bit resolution varying from 0 to 255). Reflectance values in each channel should increase linearly with light intensity (e.g., the grey color scale squares should increase in reflectance linearly). However, digital camera sensors often record reflectance in a saturating curve. This curve should be linearized before further analyses by applying a correction with linearization equations. To create these equations, you need to extract reflectance values from a subset of pictures that represents various ambient lighting conditions. If you have few pictures, you should use all of them.

In our case, we took multiple pictures for each turtle and, thus, we used the first picture taken of all individuals to cover all lighting conditions (N = 260 pictures).

Before you start working with ImageJ, prepare a spreadsheet file like this one:

	A	B	C	D	E	F	G	H	I	J	K
1	ID	No.Photo	RGB	Grey scale	Expected_sRGB	Square	Label	Area	Mean	Min	Max
2	1	5	R	Black	52	1	DSC_0522.NEF:3486-3375:Red	0.122	17.355	2	40
3	1	5	R	Neutral 3.5	85	2	DSC_0522.NEF:3693-3369:Red	0.129	36.574	22	49
4	1	5	R	Neutral 5	122	3	DSC_0522.NEF:3897-3360:Red	0.122	58.94	48	69
5	1	5	R	Neutral 6.5	160	4	DSC_0522.NEF:4098-3336:Red	0.144	86.193	75	98
6	1	5	R	Neutral 8	200	5	DSC_0522.NEF:4299-3324:Red	0.096	111.026	102	119
7	1	5	R	White	243	6	DSC_0522.NEF:4503-3306:Red	0.136	140.177	129	150
8	1	5	G	Black	52	7	DSC_0522.NEF:3486-3375:Green	0.122	16.342	9	28
9	1	5	G	Neutral 3.5	85	8	DSC_0522.NEF:3693-3369:Green	0.129	35.378	26	42
10	1	5	G	Neutral 5	122	9	DSC_0522.NEF:3897-3360:Green	0.122	56.265	49	64
11	1	5	G	Neutral 6.5	160	10	DSC_0522.NEF:4098-3336:Green	0.144	83.321	77	89
12	1	5	G	Neutral 8	200	11	DSC_0522.NEF:4299-3324:Green	0.096	107.35	101	113
13	1	5	G	White	243	12	DSC_0522.NEF:4503-3306:Green	0.136	134.341	125	140
14	1	5	B	Black	52	13	DSC_0522.NEF:3486-3375:Blue	0.122	15.8	9	26
15	1	5	B	Neutral 3.5	85	14	DSC_0522.NEF:3693-3369:Blue	0.129	34.38	26	41
16	1	5	B	Neutral 5	121	15	DSC_0522.NEF:3897-3360:Blue	0.122	54.301	45	62
17	1	5	B	Neutral 6.5	160	16	DSC_0522.NEF:4098-3336:Blue	0.144	80.024	73	86
18	1	5	B	Neutral 8	200	17	DSC_0522.NEF:4299-3324:Blue	0.096	103.279	93	110
19	1	5	B	White	242	18	DSC_0522.NEF:4503-3306:Blue	0.136	128.618	116	135
20	2	13	R	Black	52	1	DSC_0611.NEF:0582-1532:Red	0.134	21.502	0	36

For each individual (ID), you need to have three rows per grey color square from your color chart for each picture you want to analyze. The number of grey color squares can vary between color charts.

In our case, we had six grey color squares giving us 18 rows per individual (column *Square* contains the number of each row).

You will need to extract the reflectance value for each grey scale square from the color chart (in our color chart, we had six grey scales: Black, Neutral 3.5, Neutral 5, Neutral 6.5, Neutral 8, and White) for each color channel. Each grey scale square has three expected reflectance value: one per color channel (Expected_sRGB). Each row has a square number (Square) that represents the number of the region of interest that will later be created in ImageJ to calculate the reflectance value associated with each grey scale for each color channel. All other columns (e.g., Label, Area, Mean, Min and Max) contain information that will be generated by ImageJ.

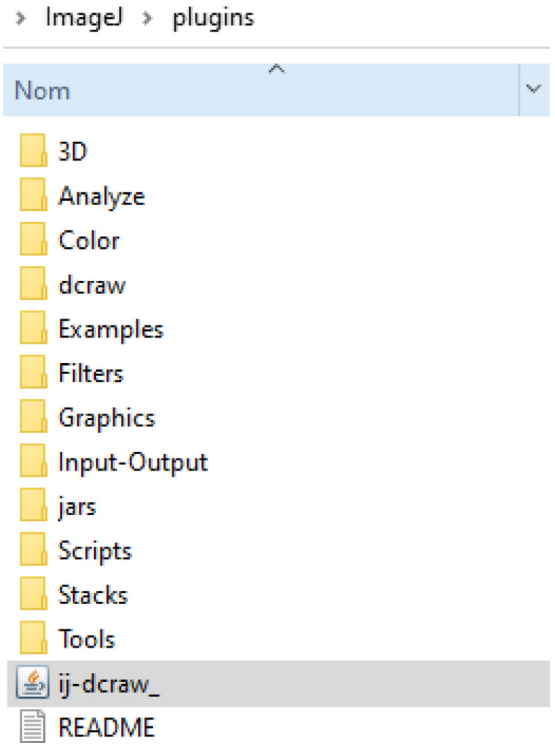
Opening a .NEF picture in ImageJ

All the steps that are performed in ImageJ are subject to slight variations due to the person performing the analyses. It is thus important to i) have the same person conduct all the analyses in ImageJ or ii) quantify the variance between measurements that were performed by different persons in ImageJ (calculate repeatability).

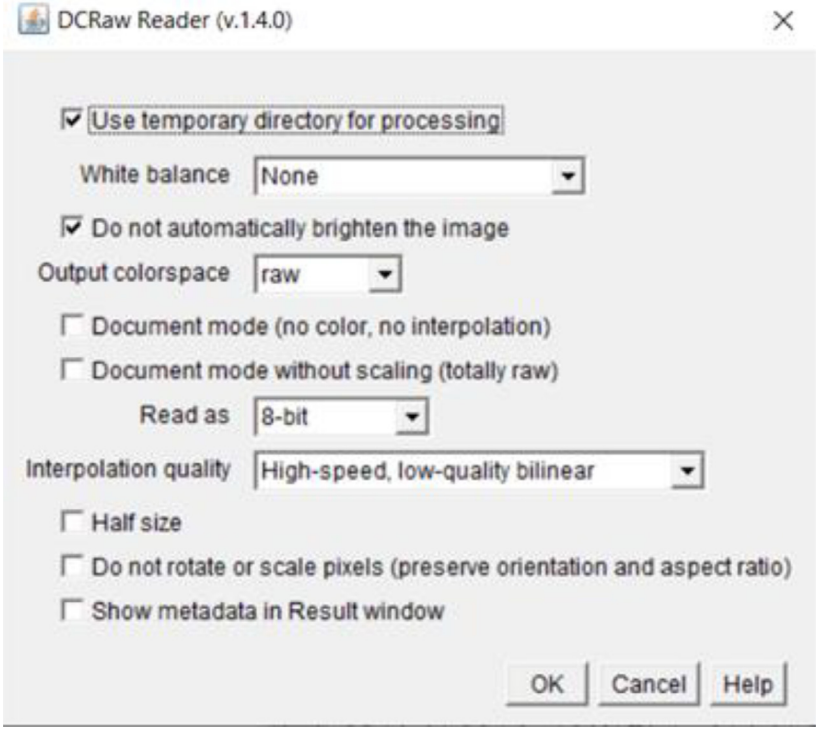
You can see steps of the next two sections (*Opening a .NEF picture in ImageJ* and *Saving pictures in Tiff format*) in this video: https://youtu.be/kbqcvgtKjmw

In our case, all pictures were saved in .NEF format, which is the equivalent of .RAW file extension for Nikon cameras. Other companies may use other RAW extensions, but they should be the same.

To open a .NEF picture in ImageJ, the DCRaw package is required (it can be found here: https://github.com/ij-plugins/ijp-dcraw; Installation instructions are provided on the same page). To install the package on your computer, you need to unzip the file and copy the executable document (ij-dcraw_.jar) in the *plugins* folder of the *ImageJ* folder on your computer.



Once the package is installed, restart ImageJ and it will now be possible to open a picture by clicking on *Pluggins >> Input-Output >> DCRaw Reader* and then selecting the appropriate file. The following menu will appear, and you can see the options we used for our analyses:



Saving pictures in Tiff format

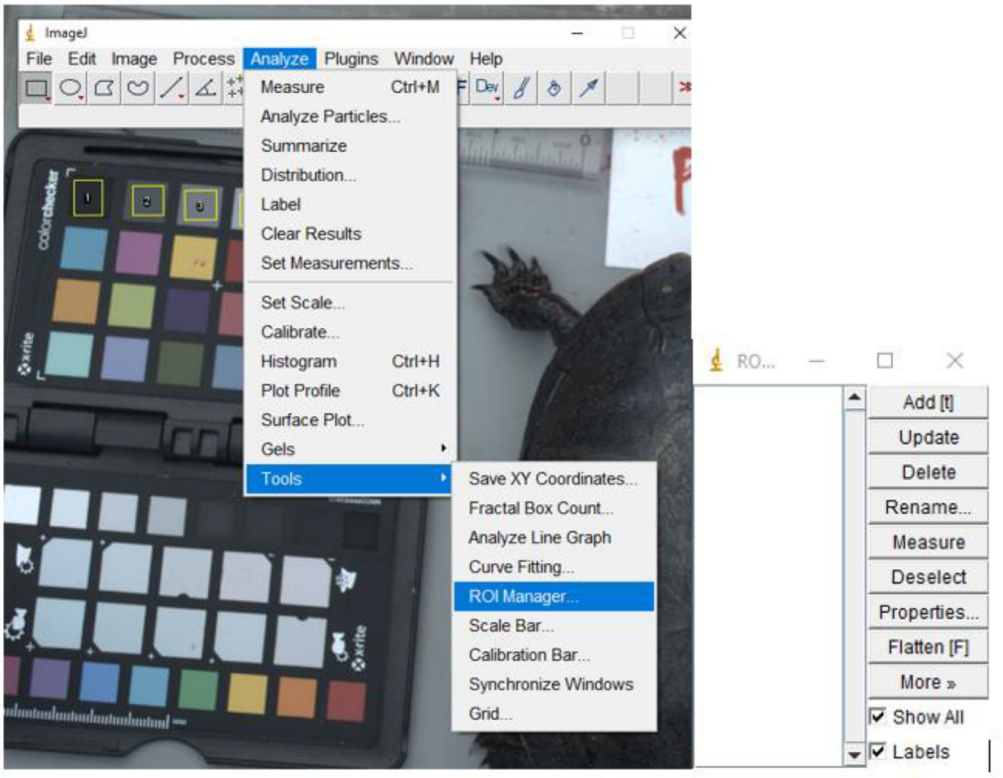
Now save your pictures as .TIFF via *File >> Save as >> Tiff*. This will allow you to work on a picture that has the parameters you previously specified instead of working on a projection of a .RAW file for which you do not know what kind of transformations have been applied. The new .TIFF picture will be the one you will use as input in the following R scripts. Even if you only use a subset of pictures to obtain the linearization equations, you must do this step for all pictures that you want to use in your final analyses.

The preview of the .RAW and .TIFF files may appear darker when viewed in ImageJ, it is normal.

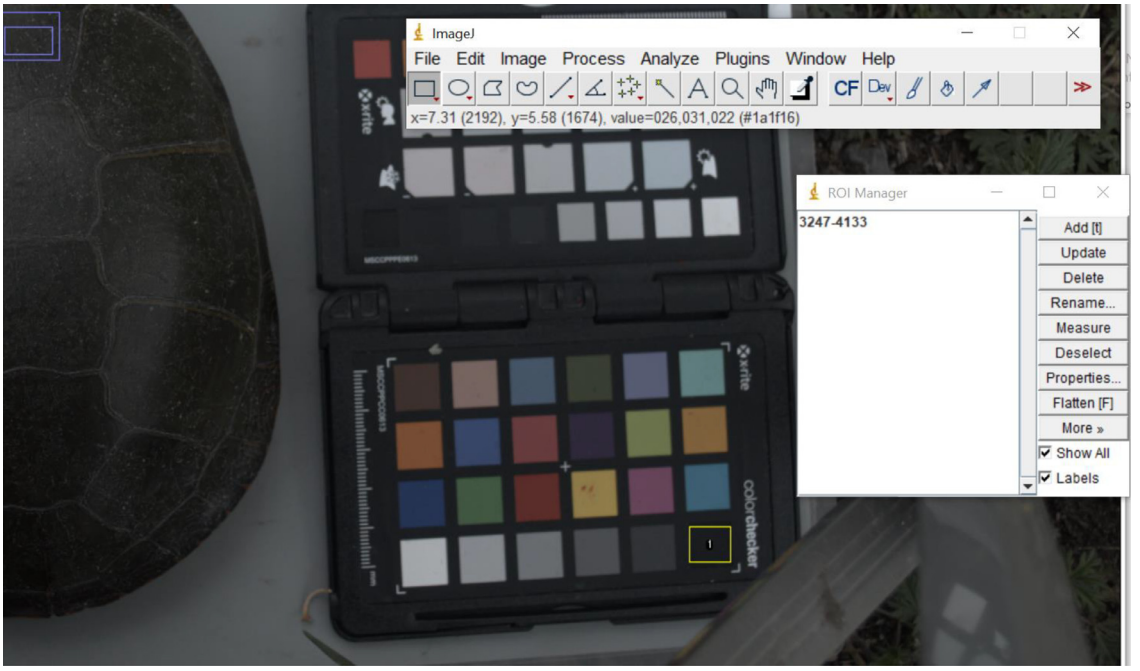
Extracting RGB scores in ImageJ

You can see steps of this section in this video: https://youtu.be/8iDWLSDFgOo

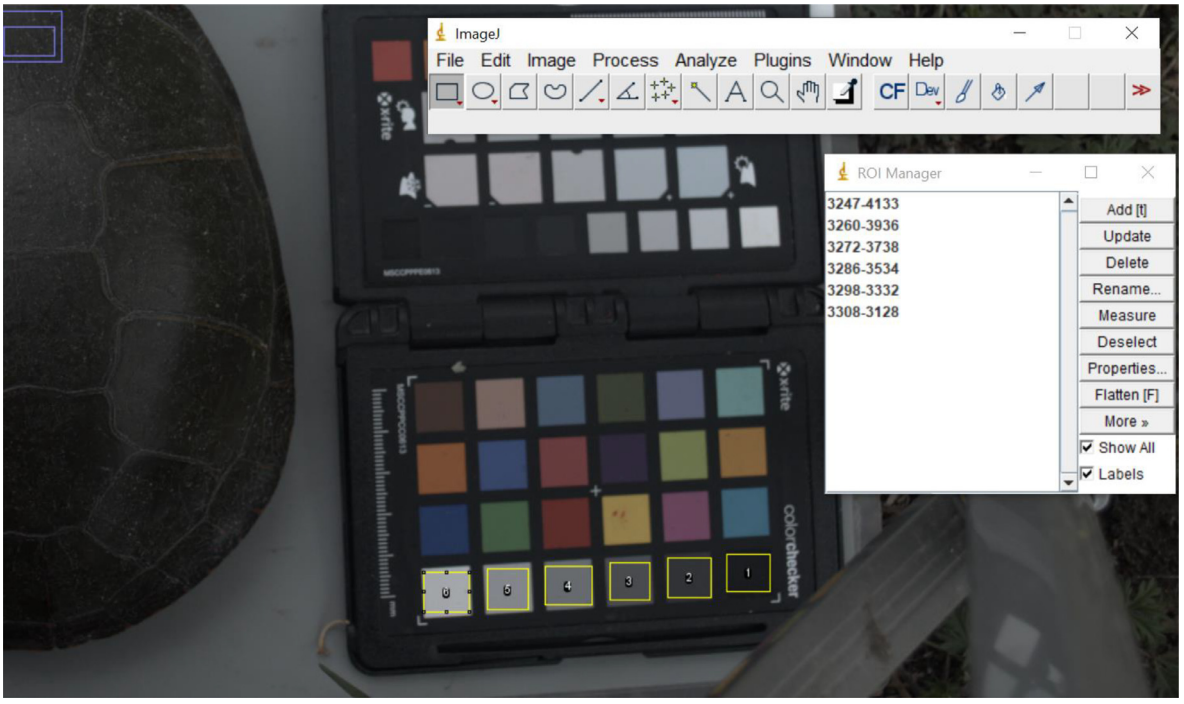
Now that you have your .TIFF picture opened in ImageJ, the first step to extract RGB scores is to create selection squares (i.e., region of interest, ROI) over the color chart's grey squares. To create multiple selections, open the *ROI Manager* by clicking on *Analyze* >> *Tools* >> *ROI Manager*. Make sure you check the *Show All* and *Labels* options in the window before you begin.



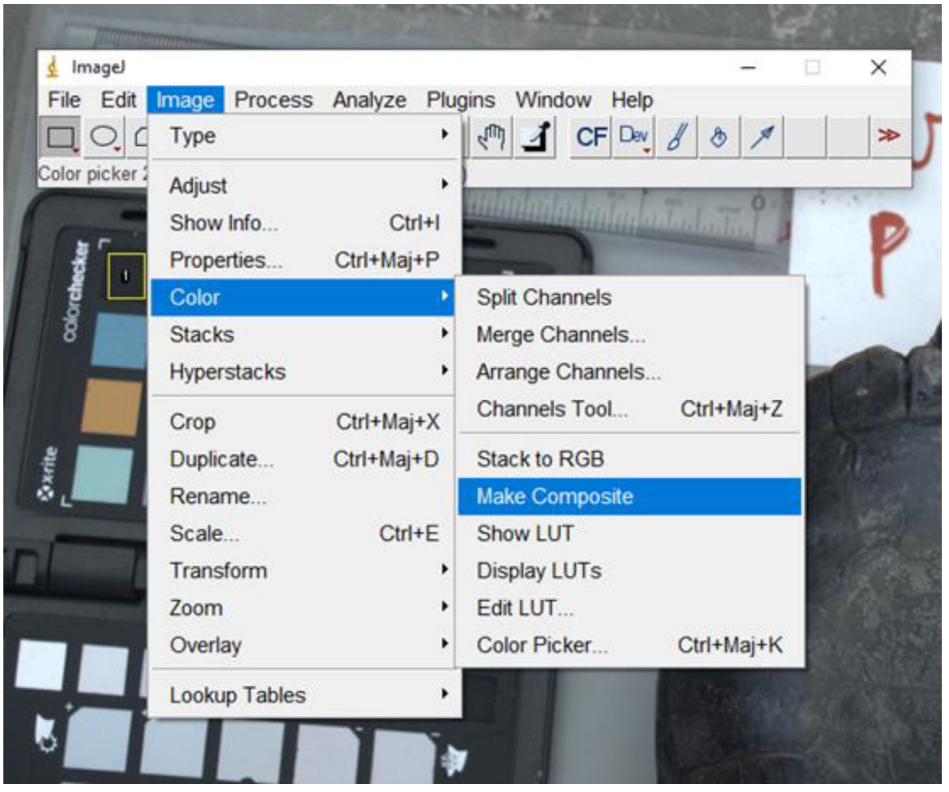
You can now draw a square over one of the chart's grey squares (make sure that your selection tool is the square). To add it to the *ROI Manager*, simply press *t* or click *Add* in the *ROI Manager*. Your selection should now appear in the *ROI Manager* as shown below:



Do the same for all grey squares making sure that you only include the color of interest (no black borders) in each square, and that all squares are approximately the same size.

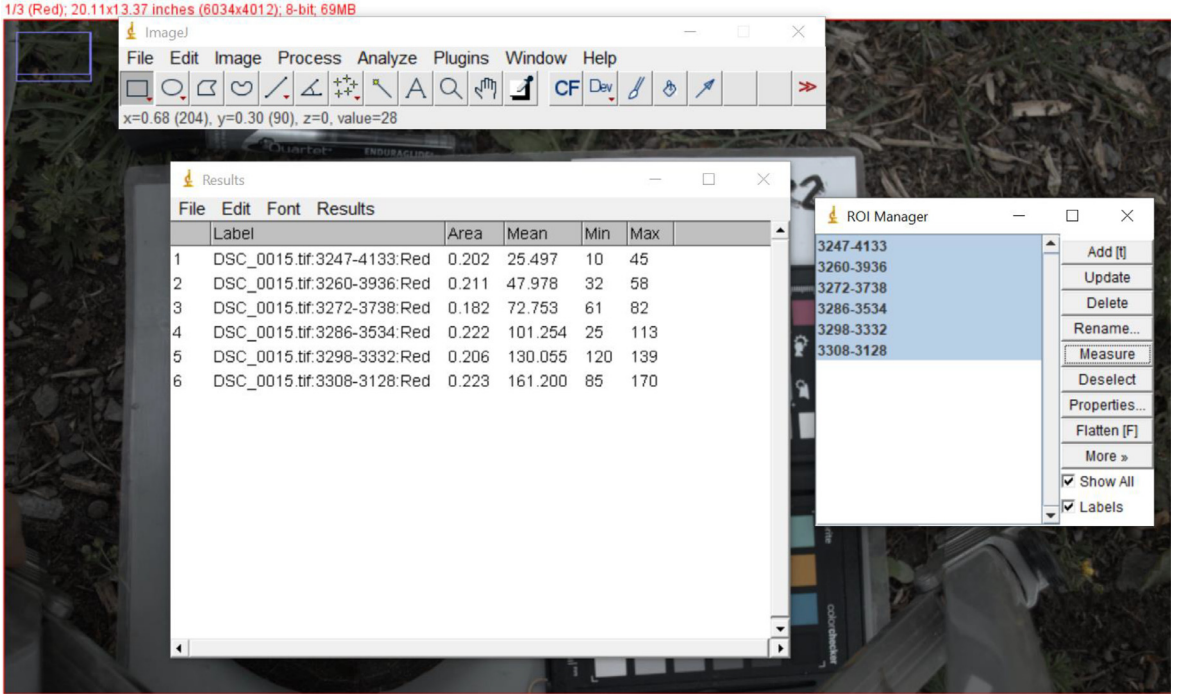


Because you need to extract the reflectance values for each of the three channels separately, you now have to convert the image into a multi-channel stack. To do so, click on *Image* >> *Color* >> *Make Composite*.

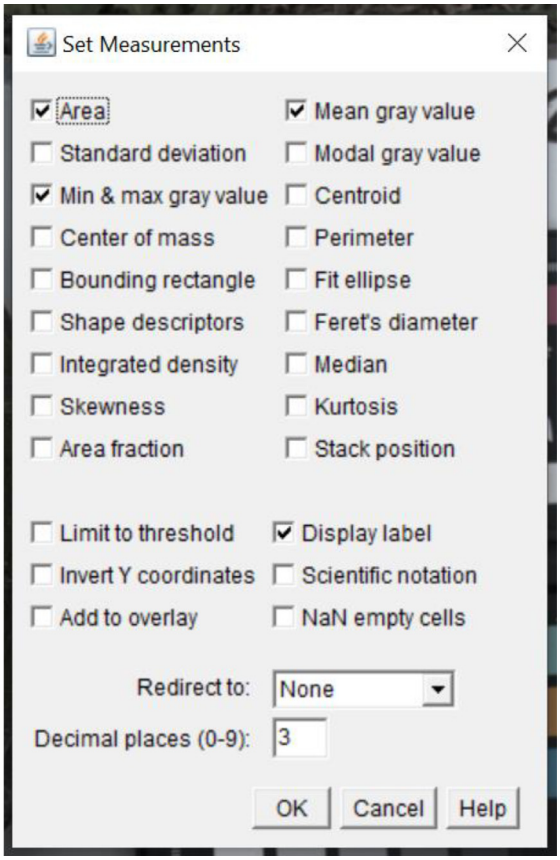


The picture should now be split in three. You can browse through the different channels by using the arrows on your keyboard or with the scrolling bar at the bottom of the picture. In the top left corner, you should see which channel is represented (Red, Green or Blue). In the picture below, you can see that we are viewing the red channel (1/3 channel):

To get the reflectance values, select all the created ROI in the *ROI Manager* and click on *Measure* in the *ROI Manager* window. This will give you a mean reflectance value (named Mean in ImageJ) for all your squares for the channel that is displayed (and other values such as the area of selection in square pixels, the min and max reflectance values in the ROI, etc.). Do the same for each of the three color channels by moving to the channel you want to analyze and click again on *Measure*. Finally, input all the measurements in your spreadsheet file.



If the labels are not displayed properly, you can go in the *Results* window, click on *Results* >> *Set* >> *Measurements* and select *Display label* >> *OK* . You will need to measure your ROIs again to activate the labeling.



Tip #1. If you use labels, add this column to your spreadsheet file because they give information about the picture and the color channel from which the measurements came from. It is also possible to save a .CSV file with those results directly in ImageJ. You can also keep the measurements of a previous picture in ImageJ and copy the results to your spreadsheet after few pictures. You just need to keep the *ROI Manager* opened and calculate the reflectance values after a few pictures.

Tip #2. To be able to quickly fill the spreadsheet file, it is best to always select the different grey color squares in the same order (i.e., always start and finish with the same ones). Make sure that the color chart is always in the same position.

Creating linearization equations

As stated in Paterson and Blouin-Demers [7], "To use photographs for the analysis of color, a digital camera's three sensors (corresponding to red, green, and blue wavelengths) should respond linearly and equally to increases in light intensity". Contrary to Paterson and Blouin-Demers [7], our observed reflectance values already appeared linearly related to expected reflectance values before any corrections because we used ImageJ's DCRaw package [3,10]. The observed reflectance values, however, were always lower than the expected reflectance values. We thus simply needed to apply a linear correction with a normal lm() function in R to obtain a better fit. We used the following equations for each color channel (i.e., camera's three sensors):

$$Q_r = a_1 + b_1 r \tag{1}$$

$$Q_g = a_2 + b_2 g \tag{2}$$

$$Q_b = a_3 + b_3 b \tag{3}$$

Where Q is the expected reflectance values from the grey color squares of the color chart, a and b are constants (for each color channel), and r, g, and b are the observed reflectance values from each color channel (red, green, and blue).

If the relationships had not been linear (as it was the case for Paterson and Blouin-Demers [7]), we could have used the following equations instead, using the nls() function in R:

$$Q_r = a_1 * \left(b_1^r\right) \tag{1a}$$

$$Q_g = a_2 * \left(b_2^g\right) \tag{2a}$$

$$Q_b = a_3 * \left(b_3^b\right) \tag{3a}$$

We used the reflectance values extracted in the previous section (Extracting reflectance values with ImageJ) and the expected reflectance values from the color chart to estimate *a* and *b* in each equation. Before going further, you need to check how your observed reflectance values are related to the expected reflectance values and, thus, determine the type of correction needed. Other equations could be used. Make sure to select the equations that are most appropriate to your data. Camera models, individual cameras, and environmental conditions may affect the response curves.

Loading R packages

Note: Here we present the packages we used in R to be able to run the following script. However, they may not all be required to conduct the analyses and some other packages may be equivalent to the ones we used. Each package needs to be installed with the function install.packages() prior to being loaded and attached in R with the function library() as shown below. We assume that you are starting with a clean R environment between each section.

```
library(readxl)
library(Hmisc)
library(tidyverse)
library(rio)
library(ggplot2)
library(ggpubr)
```

Setting the directory

To be able to run these scripts, you need to set your directory by choosing the folder where your data are saved (e.g., spreadsheet file) with the function setwd() or by following this path: *Session* >> *Set Working Directory* >> *Choose Directory*. You can also use the here package (https://here.r-lib.org/) which is less dependent on the way you organize your files.

To practice with the scripts, you can use the following Excel file available in supplementary material that contains the data we used to calculate our linearization equations: Houle-et-al_Reflectance_Values.xlsx (named *Data_Example.xlsx* in the following script). It is the same Excel file that we used in the previous section (Extracting reflectance values with ImageJ).

Importing data

As a reminder, the spreadsheet contains all observed reflectance values extracted in ImageJ from the color chart's grey squares for a subset of pictures representing various ambient lighting conditions. For each picture, the reflectance values were extracted for each color channel (Red, Green and Blue).

```
# Create a vector with the format we want for each column in our xlsx file.
Color_coltypes <- c("text", "numeric", rep("text", 2), rep("numeric", 2), "text
", rep("numeric", 4))

Color <- read_excel("Data_Example.xlsx", sheet = 2, col_types = Color_coltypes)

str(Color)

## tibble [4,698 × 11] (S3: tbl_df/tbl/data.frame)
##  $ ID           : chr [1:4698] "1" "1" "1" "1" ...
##  $ No.Photo     : num [1:4698] 5 5 5 5 5 5 5 5 5 5 ...
##  $ RGB          : chr [1:4698] "R" "R" "R" "R" ...
##  $ Grey scale   : chr [1:4698] "Black" "Neutral 3.5" "Neutral 5" "Neutral 6.
5" ...
##  $ Expected_sRGB: num [1:4698] 52 85 122 160 200 243 52 85 122 160 ...
##  $ Square       : num [1:4698] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Label        : chr [1:4698] "DSC_0522.NEF:3486-3375:Red" "DSC_0522.NEF:36
93-3369:Red" "DSC_0522.NEF:3897-3360:Red" "DSC_0522.NEF:4098-3336:Red" ...
##  $ Area         : num [1:4698] 0.122 0.129 0.122 0.144 0.096 0.136 0.122 0.1
29 0.122 0.144 ...
##  $ Mean         : num [1:4698] 17.4 36.6 58.9 86.2 111 ...
##  $ Min          : num [1:4698] 2 22 48 75 102 129 9 26 49 77 ...
##  $ Max          : num [1:4698] 40 49 69 98 119 150 28 42 64 89 ...
```

Filtering out overexposed/underexposed pictures

You should exclude all overexposed pictures (i.e., picture with a reflectance of, or near 255) to calculate your linearization equations.

In our case, we arbitrarily chose a threshold of 254.5 to identify potentially problematic overexposed pictures.

```
which(Color$Mean >= 254.5)
## integer(0)
```

You can see here that we do not have any overexposed pictures.

You also do not want underexposed pictures with a mean reflectance value very close to zero. You should remove them if it is the case.

In our case, we verified if we had pictures with a mean reflectance value of 0.5 or less.

```
which(Color$Mean <= 0.5)
## integer(0)
```

Again, you can see here that we do not have any underexposed pictures.

If you have over or underexposed pictures, you can use the code below to filter out those pictures from your dataset.

```
## To delete overexposed pictures
Color <- subset(Color, Mean <= 254.5)
## To delete underexposed pictures
Color <- subset(Color, Mean >= 0.5)
```

Transforming the reflectance values: scaling between 0 and 1

To create your linearization equations, you want your reflectance values to be relative to the highest value possible (i.e., 255). After this transformation, all your values (the expected and the observed values) will be between 0 and 1.

```
# Expected value according to the color chart
Color$Expected_sRGB_01 <- Color$Expected_sRGB / 255
# Observed mean reflectance value
Color$Mean_01 <- Color$Mean / 255
# Should be between 0 and 1
summary(Color$Expected_sRGB_01)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.2039  0.3333  0.5529  0.5630  0.7843  0.9529
```

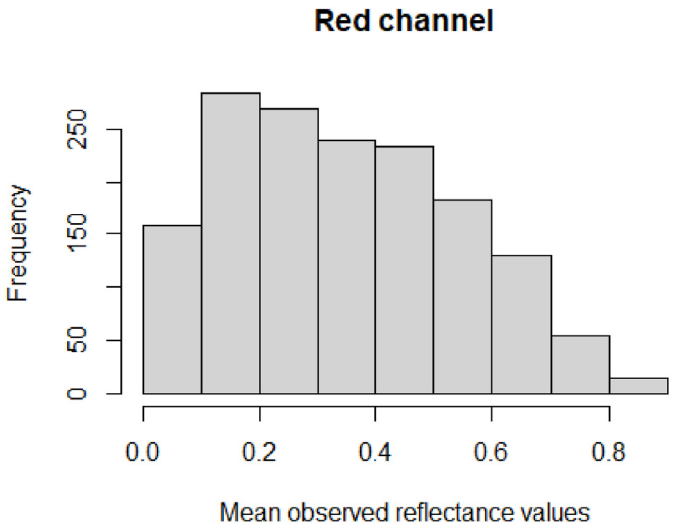
```
# Should be between 0 and 1

summary(Color$Mean_01)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.02316 0.17860 0.33293 0.34808 0.49797 0.94441
```

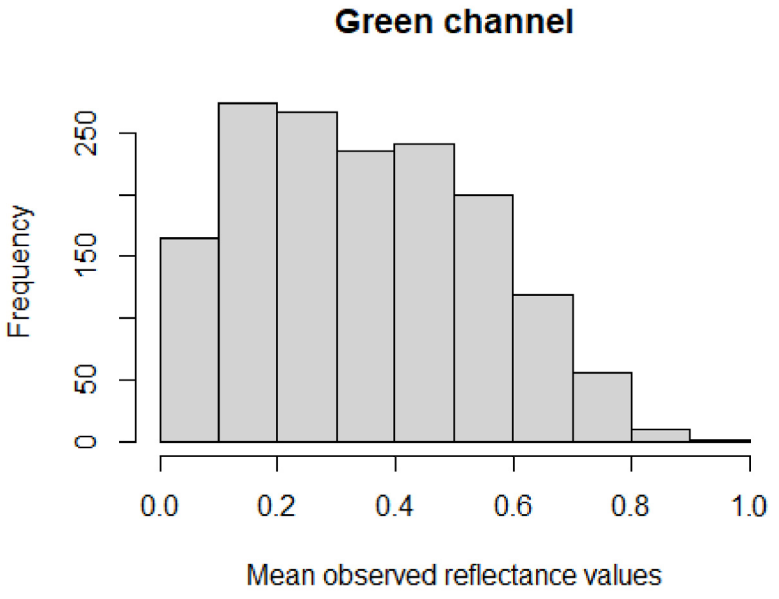
Splitting data by color channel

You need to create one linearization equation per color channel and, thus, you need to split your data per color channel.

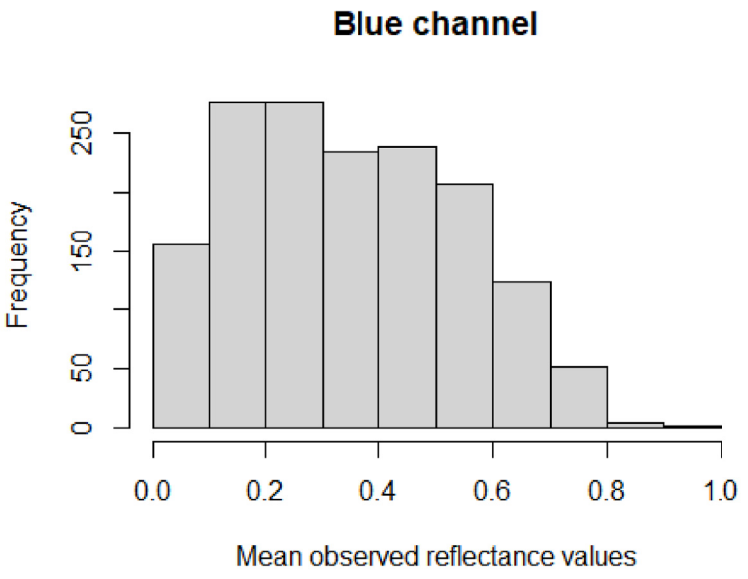
```
# Red channel
Color_R <- subset(Color, RGB == "R")
hist(Color_R$Mean_01, main = "Red channel", xlab = "Mean observed reflectance v
alues")
```




```
# Green channel
Color_G <- subset(Color, RGB == "G")
hist(Color_G$Mean_01, main = "Green channel", xlab = "Mean observed reflectance
values")
```



```
# Blue channel
Color_B <- subset(Color, RGB == "B")
hist(Color_B$Mean_01, main = "Blue channel", xlab = "Mean observed reflectance
values")
```



Linearization equations

Now, you can create a linear model for each color channel and extract the slope and the intercept of each model to build your equations.

In our case, we used the lm() function for our equations as mentioned earlier.

```
redlinear <- lm(Expected_sRGB_01 ~ Mean_01, data = Color_R)
summary(redlinear)

##
## Call:
## lm(formula = Expected_sRGB_01 ~ Mean_01, data = Color_R)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.27288 -0.06205 -0.01210  0.05449  0.43465
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.141803   0.004648   30.51   <2e-16 ***
## Mean_01     1.213138   0.011629  104.32   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0909 on 1564 degrees of freedom
## Multiple R-squared:  0.8744, Adjusted R-squared:  0.8743
## F-statistic: 1.088e+04 on 1 and 1564 DF,  p-value: < 2.2e-16
```

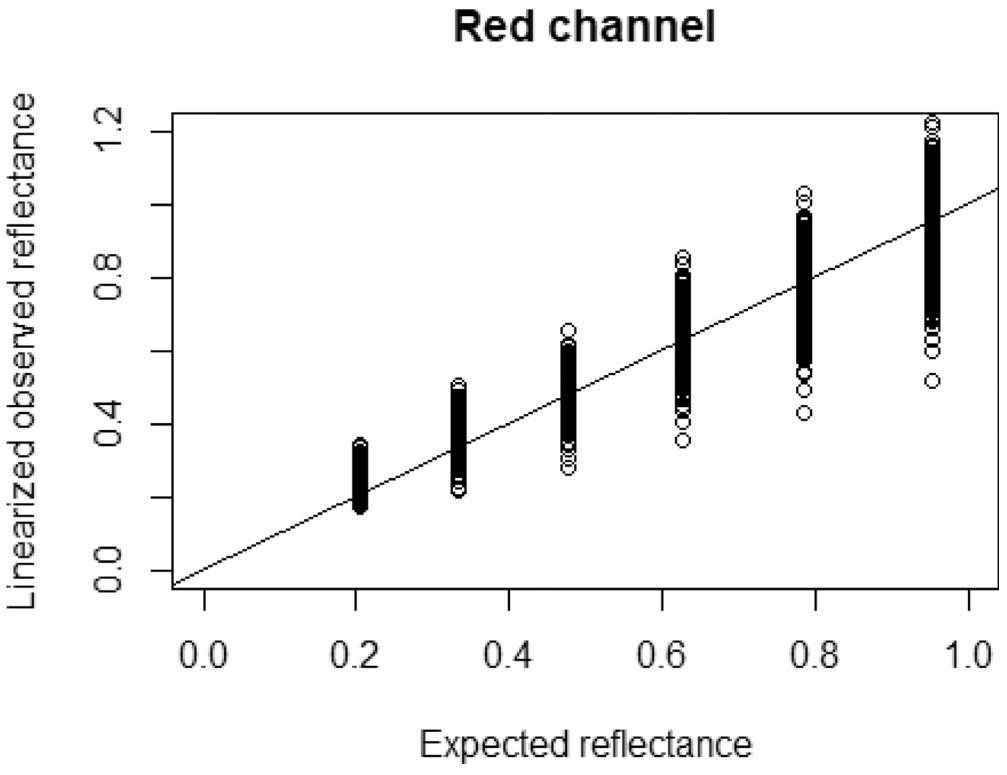
The intercept is 0.141803 and the slope is 1.213138 for the linearization equation of the red channel.

Now, you can linearize the mean observed reflectance values for the red channel with this equation that contains the constants that were extracted from the model.

```
r.2.linear <- 0.141803 + (Color_R$Mean_01 * 1.213138)
```

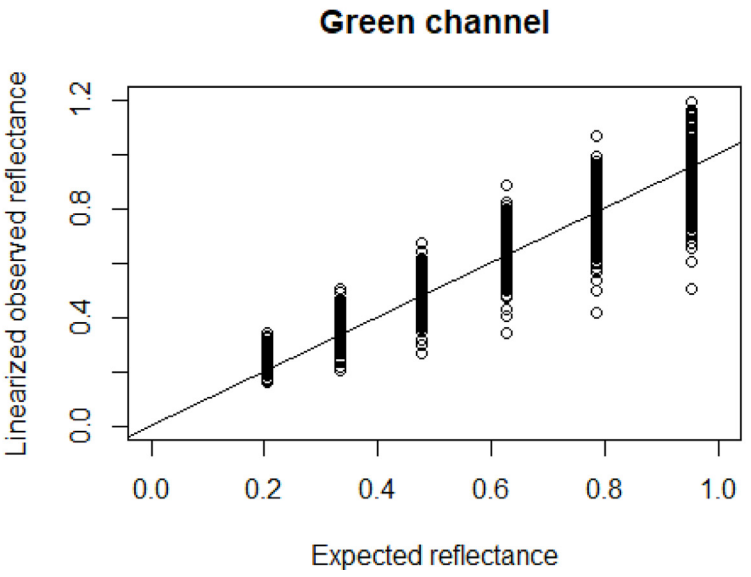
You can visualize the relationship between the linearized observed values and the expected reflectance values from the color chart. If the linearization equation works well, the data are supposed to follow the black line (1:1).

```
plot(r.2.linear ~ Color_R$Expected_sRGB_01, main = "Red channel", xlab = "Expec
ted reflectance", ylab = "Linearized observed reflectance", xlim = c(0, 1), yli
m = c(0, 1.2))
abline(0, 1)
```

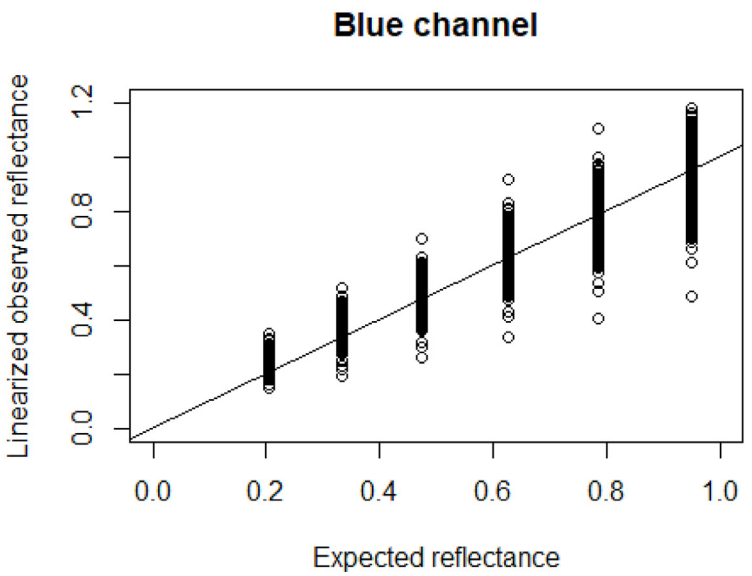


Repeat the same steps for the green and blue channels.

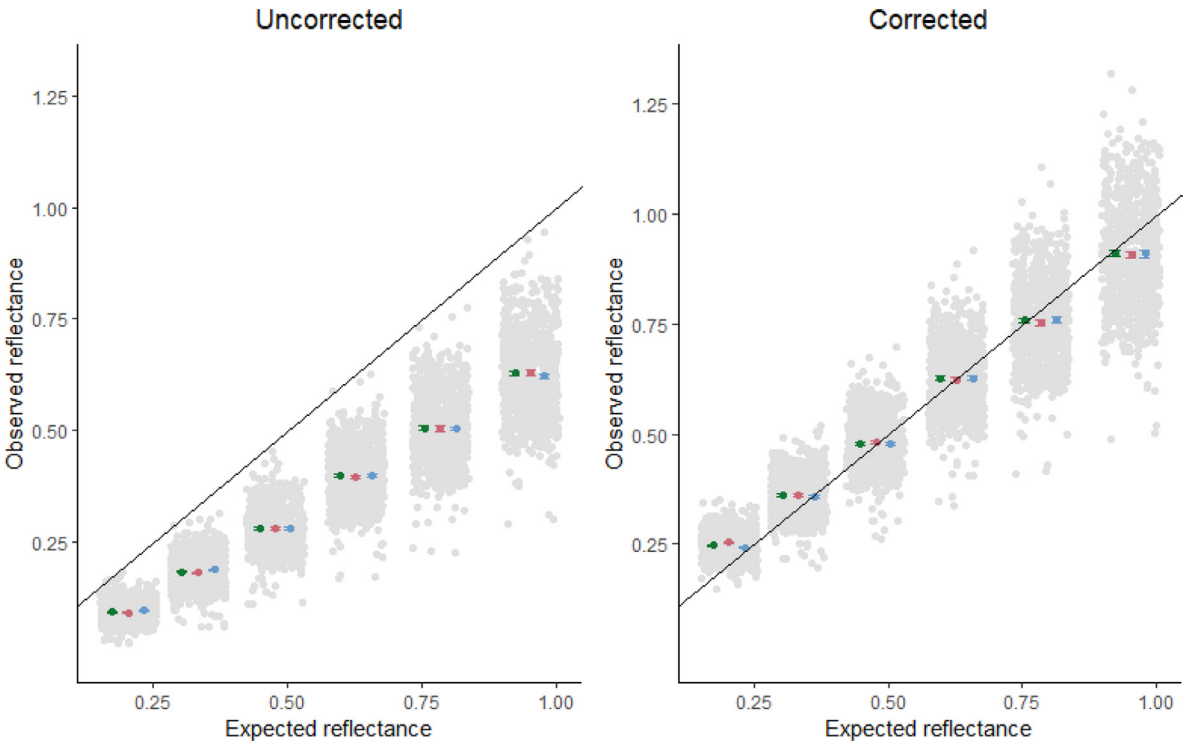
```
g.2.linear <- 0.130617 + (Color_G$Mean_01 * 1.242959)
```




```
b.2.linear <- 0.118825 + (Color_B$Mean_01 * 1.271826)
```



You can visualize the changes made by the linearization equations to the observed reflectance values from the three channels.



The figures depict the relationship between the expected reflectance values and the observed reflectance values (Left: uncorrected; Right: corrected) for the three color channels (Red, Green and Blue: each color dot depicts the mean ± SE of their respective color channel). Each gray dot represents one unique observation. You can see that the relationship between the expected and the corrected observed reflectance values follows better the reference line (1:1). This confirms that the linearization equations are working.

Equalizing the channels

The grey scales of a color chart are designed to reflect equally across red, green, and blue. However, camera sensors sometimes have different sensitivities and a correction to equalize color channels is necessary. In some cases, the linearization process will also equalize the three color channels (e.g., transformed R = transformed G = transformed B). You should thus confirm that linearized pixel scores of grey color scales from the color chart are equal in each channel using paired *t*-tests. If it is not the case, see Stevens et al. [6].

```
t.test(r.2.linear, g.2.linear, paired = TRUE)

##
##  Paired t-test
##
## data:  r.2.linear and g.2.linear
## t = -0.0011145, df = 1565, p-value = 0.9991
## alternative hypothesis: true mean difference is not equal to 0
## 95 percent confidence interval:
##  -0.0008726124  0.0008716213
## sample estimates:
## mean difference
##   -4.955514e-07

t.test(r.2.linear, b.2.linear, paired = TRUE)

##
##  Paired t-test
##
## data:  r.2.linear and b.2.linear
## t = 1.5925, df = 1565, p-value = 0.1115
## alternative hypothesis: true mean difference is not equal to 0
## 95 percent confidence interval:
##  -0.0003029215  0.0029174415
## sample estimates:
## mean difference
##      0.00130726

t.test(g.2.linear, b.2.linear, paired = TRUE)

##
##  Paired t-test
##
## data:  g.2.linear and b.2.linear
## t = 3.2623, df = 1565, p-value = 0.001129
## alternative hypothesis: true mean difference is not equal to 0
## 95 percent confidence interval:
##  0.0005214506 0.0020940605
## sample estimates:
## mean difference
##     0.001307756
```

24

You need to correct your p value according to the number of comparisons made (three in this case). We choose the conservative Bonferroni correction:

$$\alpha = \frac{0.05}{3} = 0.0167$$

In our case, we only observed a significant difference between the green and blue channels. The mean difference, however, was very small (0.001). We thus concluded that the linearized pixel scores of the grey color scales were equal in each color channel (Red, Green and Blue).

Linearizing pictures

Now that the constants *a* and *b* are known, it is possible to linearize all your pictures (not just the subsample of pictures previously used) with the equations created in the previous section (Creating linearization equations).

In our case, our final dataset included two .TIFF pictures per individual, one dorsal photo of the top of the head (usually showing two yellow spots) and one ventral picture (showing both yellow and red on the neck). Here is an example of a set of pictures for one turtle:

Before running this script, make sure that all your pictures are saved in .TIFF format and have the same properties.

Loading R packages

See *Loading R packages* note in *Creating linearization equations* section.

```
library(imageviewer)
library(magick)
library(rsvg)
library(beepr)
library(hexView)
library(tiff)
library(EBImage)

# You can use this setting to install packages (e.g., EBImage) from other repos
itories than CRAN (e.g., BiocManager) with the function install.packages().
options(repos = BiocManager::repositories())
```

The installation of EBImage should work if you follow these steps:

https://www.bioconductor.org/packages/devel/bioc/vignettes/EBImage/inst/doc/EBImage-introduction.html

If you encounter difficulties for the installation, you can try installing it with RStudio in administrator mode.

Setting the directory

To be able to run these scripts, you need to set your directory by choosing the folder where your pictures (in .TIFF) are saved with the function setwd() or by following this path: *Session* >> *Set Working Directory* >> *Choose Directory*. You can also use the here package (https://here.r-lib.org/) which is less dependent on the way you organize your files.

Creating a list of pictures

You need to create a list of pictures to work on (put all pictures to be linearized in the appropriate directory/folder). Subfolders are needed if there are several pictures to process or if you have limited computing power (e.g., R may run out of memory).

In our case, we were able to process about 10 pictures at a time.

```
in.images <- list.files(path = "images", pattern = "DSC", full.names = FALSE, r
ecursive = TRUE)
in.images

## [1] "DSC_0022.tif" "DSC_0023.tif"
```

This step allows the creation of an object containing a list of all the pictures found in the selected subfolder (path = "path to the folder where a subset of pictures is located", and recursive = TRUE allows access to the subfolders inside the directory selected). You do not need to write the entire path given that the directory is already set. The pattern argument indicates a common pattern of names in all pictures you want to open.

In our case, we had a folder named *images* in our directory. All our pictures started with *DSC*, which was used as the pattern argument. This tutorial's subfolder only includes two pictures: DSC_0022 and DSC_0023. All steps in the following sections will be performed on these two pictures.

Converting pictures to arrays

Creating a function that formats pictures correctly

When a picture is uploaded in R, the different color channels are not automatically accessible. Instead, the picture is represented as one matrix. Since you need to apply your linearization equations to the three channels of your pictures (Red, Green and Blue), you need to access each color channel's array.

This function allows you to do so:

```
format.image <- function(y) {
  # creates an empty list
list_array <- list()
  for (n in 1:length(y)) {
    # transforms the name of the pictures
    image <- paste("Array", y[n], sep = "")
    # read the .TIFF file
    y2 <- image_read(y[n])
    print(dim(y2))
    # this step can be activated by deleting the (#). This step allows you to s
ee the picture in R-studio viewer (but it will take more time)
    # print(y2)
    print(image_info(y2))
    # converts to an array to be able to change the color values
    y_array <- as.integer(y2[[1]])
    y_array <- transpose(y_array)
    y_array <- y_array / 255
    print(y_array)
    list_array[[image]] <- y_array
  }
  return(list_array)
}
```

Applying the previously created function to the pictures

```
picture_list <- format.image(in.images)
```

Make sure you can access the elements inside the list.

In our case, we confirmed that the previous function worked by running the following code line, which looks at the reflectance values of picture #1 of our list between pixel rows 5 and 10, and pixel columns 10 and 20 for the third matrix (blue). The first matrix is the red channel, the second is the green channel and the third one is the blue channel.

```
picture_list[[1]][5:10, 10:20, 3]
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 0.1490196 0.1490196 0.1490196 0.1411765 0.1333333 0.1333333 0.1372549
## [2,] 0.1529412 0.1529412 0.1490196 0.1411765 0.1333333 0.1372549 0.1372549
## [3,] 0.1529412 0.1529412 0.1490196 0.1450980 0.1372549 0.1372549 0.1372549
## [4,] 0.1529412 0.1529412 0.1490196 0.1450980 0.1450980 0.1372549 0.1333333
## [5,] 0.1490196 0.1490196 0.1490196 0.1450980 0.1450980 0.1411765 0.1372549
## [6,] 0.1450980 0.1450980 0.1490196 0.1450980 0.1450980 0.1450980 0.1450980
##           [,8]      [,9]     [,10]     [,11]
## [1,] 0.1450980 0.1568627 0.1568627 0.1607843
## [2,] 0.1529412 0.1686275 0.1607843 0.1490196
## [3,] 0.1490196 0.1607843 0.1529412 0.1490196
## [4,] 0.1411765 0.1490196 0.1490196 0.1490196
## [5,] 0.1411765 0.1411765 0.1372549 0.1333333
## [6,] 0.1372549 0.1333333 0.1254902 0.1176471
```

We also checked if we could see the dimensions of picture #1 in our list.

```
dim(picture_list[[1]])
```

```
## [1] 6034 4012    3
```

You can see the width (6034 pixels), the height (4012 pixels), and the three matrices (3: Red, Green, Blue) of picture #1. In addition, you can verify that the reflectance values are indeed between 0 and 1.

```
summary(picture_list[[1]])
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.1020  0.1686  0.2003  0.2667  0.8314
```

You can also use the following function to see all reflectance values from all pixels of picture #1. The output is not shown here because it is too large.

```
picture_list[1]
```

Applying the linearization equations

Here are the equations we obtained in the previous section (Creating linearization equations):

```
# function to change RED pixel values
fred = function(x) {0.141803 + (1.213138 * x)}
# function to change GREEN pixel values
fgreen = function(x) {0.130617 + (1.242959 * x)}
# function to change BLUE pixel values
fblue = function(x) {0.118825 + (1.271826 * x)}
```

We created a function that will apply the different equations to their respective matrix. In other words, the function will allow you to apply the linearization equation of the red channel to the red matrix, the green equation to the green matrix, and the blue equation to the blue matrix.

```
change.image <- function(x, output_path) {
   for(m in 1:length(x)){
    # apply the red linearization function to the red channel
    r <- apply(x[[m]][500:4500, 500:4000, 1], MARGIN = c(1, 2), FUN = fred)
    # apply the green linearization function to the green channel
    g <- apply(x[[m]][500:4500, 500:4000, 2], MARGIN = c(1, 2), FUN = fgreen)
    # apply the blue linearization function to the blue channel
    b <- apply(x[[m]][500:4500, 500:4000, 3], MARGIN = c(1, 2), FUN = fblue)
    # this step re-stacks the three matrices in an array. The order of the thre
e layers is important here: red, green and blue.
    z <- Image(array(dim = c(nrow(r), ncol(r), 3), data = cbind(r, g, b)))
    # set to color mode
    colorMode(z) <- Color
    # the new picture will use the origin name, but will add "_linearized" at t
he end and will be saved in the .TIFF format
    output_filename <- paste(names(x[m]), "_linearized", ".TIFF", sep = "")
    # specify the path where the linearized pictures will be saved
    output_fullpath <- file.path(output_path, output_filename)
    writeImage(z, output_fullpath)
  }
}
```

Here, the pictures from the picture list are in their original dimensions and are cropped by the change.image() function by selecting a subset of pixels. However, cropping the pictures beforehand could reduce computing time during the linearization process, especially if the zone of interest is relatively small compared to the rest of the picture.

In our case, we did not crop the pictures prior to running the script. The cropping was done via the change.image() function. Thus, the resulting pictures are linearized and only include the area located between pixel rows 500 and 4500, and between pixel columns 500 and 4000 for the three matrices. This helps reduce computing time. If you decide to crop your pictures this way, you need to make sure that all pictures still include the whole area of interest, as well as the ruler (or at least part of it).

You should save all your linearized pictures in a new subfolder (e.g., subfolder named “images_linearized”). The change.image() function allows you to specify the folder path where you want to save your linearized pictures.

Now, you can apply the function to the list of pictures:

```
output_folder <- file.path("X:", "Your", "file", "Path", fsep="/")

change.image(picture_list, output_folder)
```

This step can take several minutes.

In our case, the linearization of two pictures took about 5 min on a standard laptop with an Intel® Core™ i5–5300 U processor with 16 Go of memory. Once it is done, the newly linearized pictures (named ArrayDSC_0022.tif_linearized and ArrayDSC_0023.tif_linearized) should be saved in the folder you specified in the path.

Rescaling pictures and selecting the area of interest

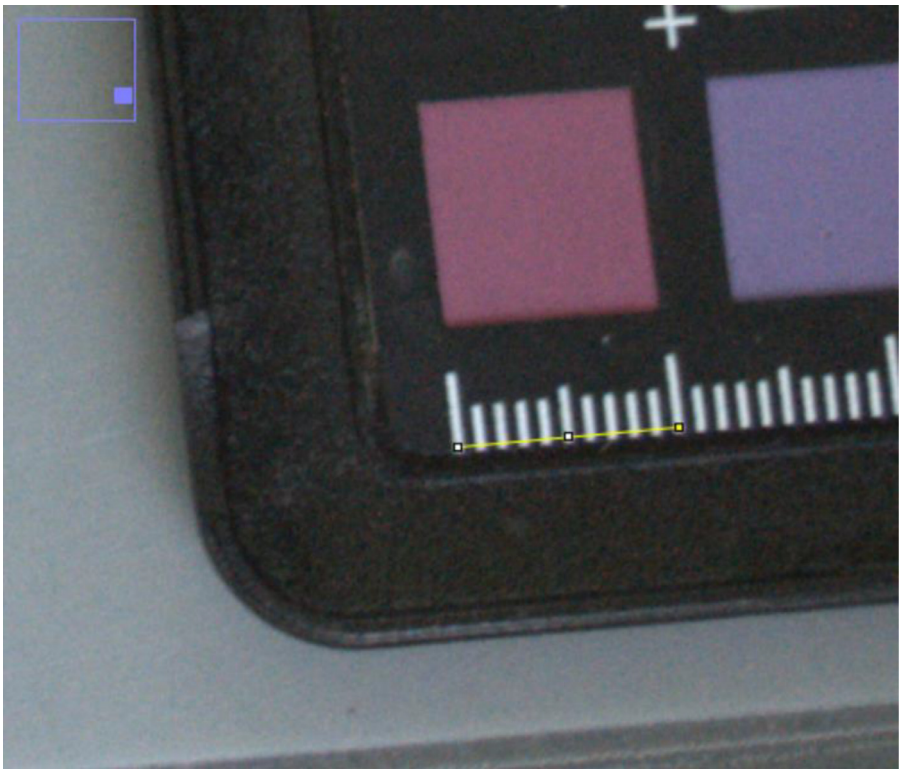
Rescaling pictures

Before going further, it is necessary to scale all pictures similarly. By doing this, you are ensuring that lengths and areas (in pixels) measured in one picture are comparable to that measured in other pictures. By using a known length item, here a ruler within the color chart, we can estimate the relationship between a fixed length in cm with a length in pixels.

In ImageJ, you can scale the pictures previously linearized and cropped in the previous section (Linearizing pictures). In the following section, we used a different head picture than in the previous section. The picture is also available in supplemental material (Picture_Section5.tiff).

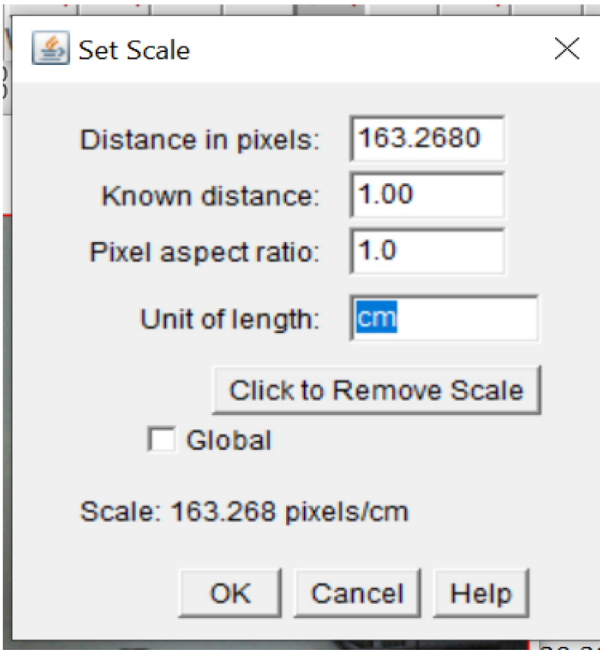
You can see the steps of this section in this video: https://youtu.be/IzjauL99_7Q.

First, you need to measure a line of known length (i.e., a section of the ruler included in all pictures) with the straight-line tool.



Then, in *Analyze >> Set Scale*, the measured distance should appear in the first box (i.e., *Distance in pixels*). The *Known distance* should be changed to the appropriate value.

In our case, we used a known distance of 1 cm. We thus changed *Known distance* to 1.00 and the *Unit of length* to cm.



The scaling factor of the picture (i.e., Distance in pixels: 163.2680) needs to be noted in a new column of your spreadsheet file. It is also possible to create a new spreadsheet file with all the pictures that will be analyzed if you only analyzed a subset of your pictures to create the linearization equations.

In our case, we added one column to enter the scaling factor (i.e., Scaling_factor_(pixels/cm)). We also added another column to compute the number of pixels in 10 cm (i.e., Number_of_pixels_in_10_cm). This column will later be used to uniformize the background size of all pictures (10 × 10 cm; see the sub-section *Resizing pictures on a 10 cm X 10 cm background* of this section). This step is necessary because, to our knowledge, ImageJ only allows background rescaling in pixel units (and not in cm).

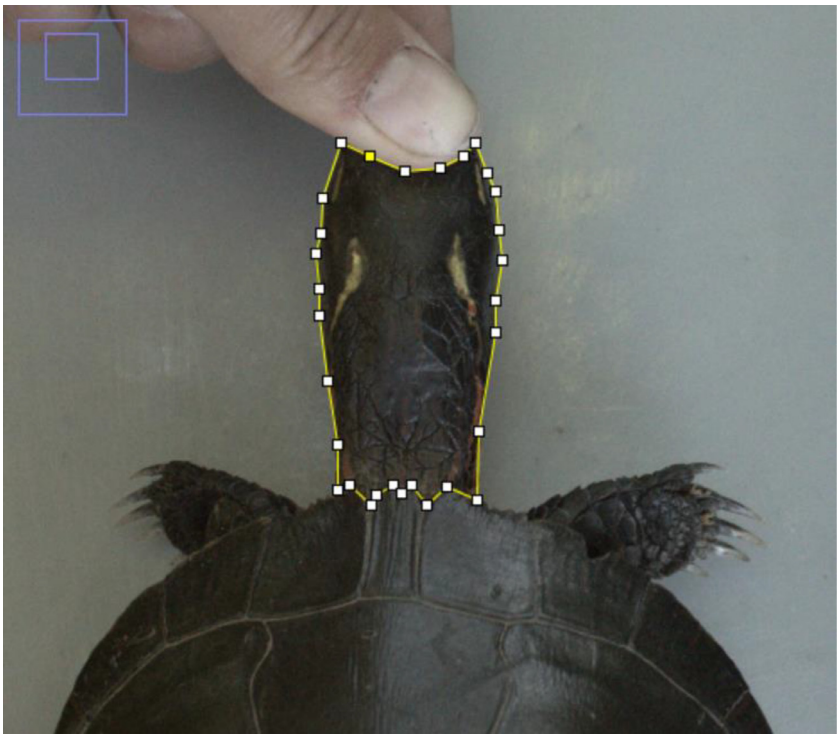
	A	B	C	D	E	F	G	H	I	J	K	L	M
1	ID	No.Photo	RGB	Grey scale	Expected_sRGB	Square	Label	Area	Mean	Min	Max	Scaling_factor_(pixels/cm)	Number_of_pixels_in_10_cm
2	1	5	R	Black	52	1	DSC_0522.NEF:3486-3375:Red	0.122	17.355	2	40	163.2680	1632.68
3	1	5	R	Neutral 3.5	85	2	DSC_0522.NEF:3693-3369:Red	0.129	36.574	22	49	163.2680	1632.68
4	1	5	R	Neutral 5	122	3	DSC_0522.NEF:3897-3360:Red	0.122	58.94	48	69	163.2680	1632.68
5	1	5	R	Neutral 6.5	160	4	DSC_0522.NEF:4098-3336:Red	0.144	86.193	75	98	163.2680	1632.68
6	1	5	R	Neutral 8	200	5	DSC_0522.NEF:4299-3324:Red	0.096	111.026	102	119	163.2680	1632.68
7	1	5	R	White	243	6	DSC_0522.NEF:4503-3306:Red	0.136	140.177	129	150	163.2680	1632.68
8	1	5	G	Black	52	7	DSC_0522.NEF:3486-3375:Green	0.122	16.342	9	28	163.2680	1632.68
9	1	5	G	Neutral 3.5	85	8	DSC_0522.NEF:3693-3369:Green	0.129	35.378	26	42	163.2680	1632.68
10	1	5	G	Neutral 5	122	9	DSC_0522.NEF:3897-3360:Green	0.122	56.265	49	64	163.2680	1632.68
11	1	5	G	Neutral 6.5	160	10	DSC_0522.NEF:4098-3336:Green	0.144	83.321	77	89	163.2680	1632.68
12	1	5	G	Neutral 8	200	11	DSC_0522.NEF:4299-3324:Green	0.096	107.35	101	113	163.2680	1632.68
13	1	5	G	White	243	12	DSC_0522.NEF:4503-3306:Green	0.136	134.341	125	140	163.2680	1632.68
14	1	5	B	Black	52	13	DSC_0522.NEF:3486-3375:Blue	0.122	15.8	9	26	163.2680	1632.68
15	1	5	B	Neutral 3.5	85	14	DSC_0522.NEF:3693-3369:Blue	0.129	34.38	26	41	163.2680	1632.68
16	1	5	B	Neutral 5	121	15	DSC_0522.NEF:3897-3360:Blue	0.122	54.301	45	62	163.2680	1632.68
17	1	5	B	Neutral 6.5	160	16	DSC_0522.NEF:4098-3336:Blue	0.144	80.024	73	86	163.2680	1632.68
18	1	5	B	Neutral 8	200	17	DSC_0522.NEF:4299-3324:Blue	0.096	103.279	93	110	163.2680	1632.68
19	1	5	B	White	242	18	DSC_0522.NEF:4503-3306:Blue	0.136	128.618	116	135	163.2680	1632.68
20	2	13	R	Black	52	1	DSC_0611.NEF:0582-1532:Red	0.134	21.502	0	36		

Note: To determine how many pixels were needed on the background so that it would work with all pictures, we first used a subset of 50 pictures. This step confirmed that a background of 10 × 10 cm was sufficient to avoid cropping any important portion of our pictures. With the spreadsheet file, we then computed the number of pixels that corresponded to 10 cm (e.g., 163.268 pixels/cm * 10 cm = 1632.68). Thus, the size of your background will depend on the size of your area of interest and on the size variation of this area between pictures.

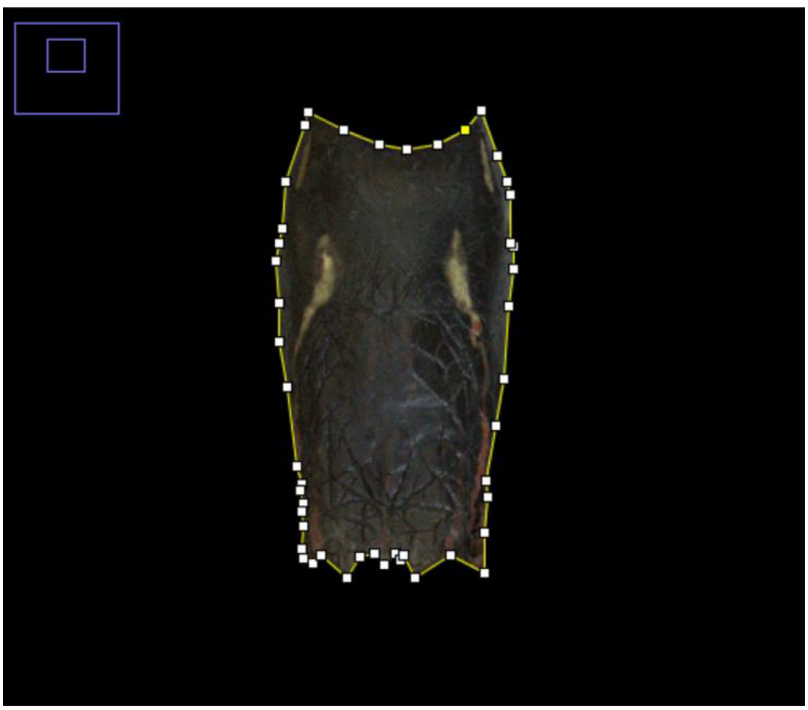
Selecting the area of interest

You can delimit the area of interest with the freehand selection tool. The selection can then be adjusted by selecting *Edit* >> *Selection* >> *Convex Hull*. You can create a new knot by pressing *Shift* and clicking on a knot. If you want to delete a knot, you can do so by pressing *Ctrl* and clicking on a knot.

Here is an example of the area of interest, selected with the freehand selection tool and adjusted manually afterwards:



Then, you can erase everything outside of your selection by using *Edit* >> *Clear Outside*

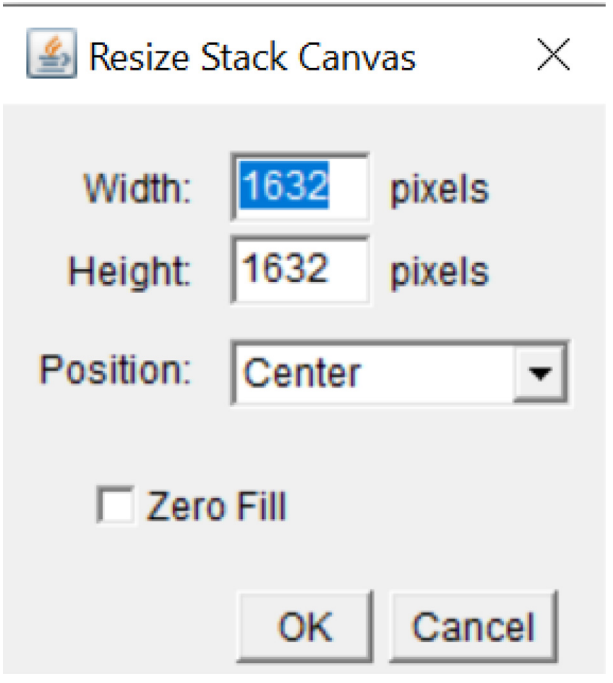


You can then crop the picture by selecting *Image* >> *Crop*

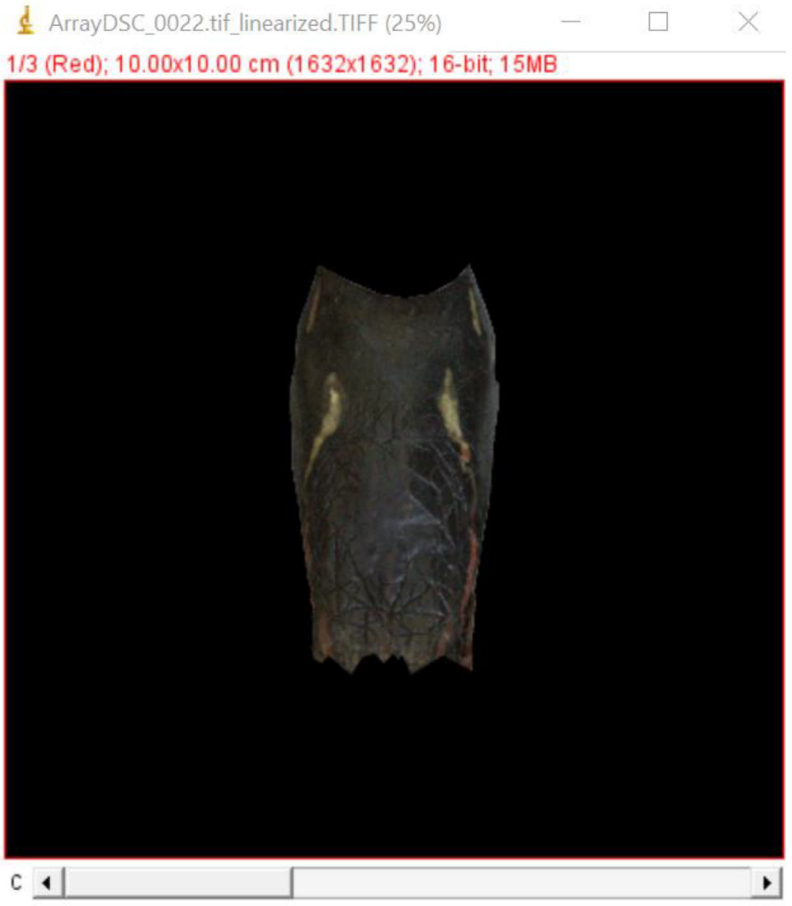
Resizing pictures on a 10 cm x 10 cm background

Lastly, you can resize the black background to make it uniform between pictures by clicking on *Image* >> *Adjust* >> *Canvas size*. You will need to fill the width and height values with the ones you computed in the *Number_of_pixels_in_10_cm* column of your spreadsheet file for this specific picture.

In our case, canvas width and height were changed to 1632.68 pixels:



Now that the picture is scaled and resized, it should look like this:



The picture is ready to be used in the following steps. We recommend saving your modifications as a new picture with a file name including as much information as possible.

In our case, the following information was included in the picture's name:

- The individual's ID;
- The first letter of Head or Neck (to indicate the region of interest in the picture);
- The mention that the picture was linearized, scaled and cropped, and that the background was uniformized to 10 cm.

For example, AB_H_linearized_scaled_cut_10.tif, for the head picture of the turtle identified as AB.

Classifying pictures before standardization & binarization

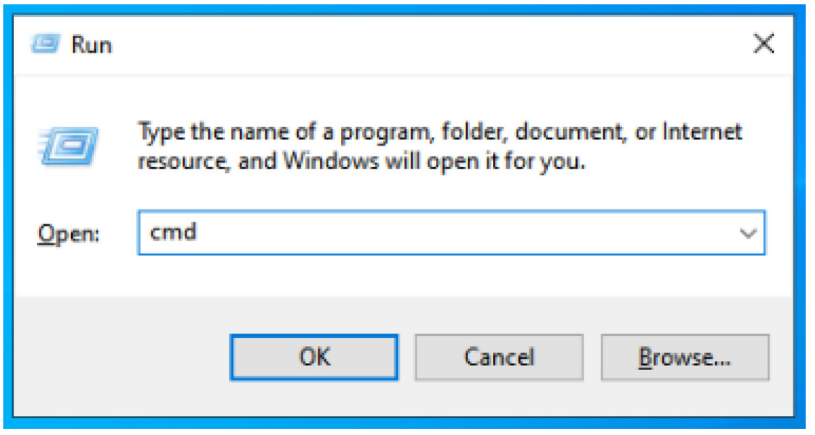
By now, all your pictures are linearized, scaled, resized, and saved in the same folder on your computer. Before proceeding to the standardization/binarization, you need to group all your cropped and linearized pictures in different folders according to the area of interest that you want to analyze (if you have multiple areas). These folders will then be used in the following R scripts.

In our case, all head pictures were grouped in one folder and all neck pictures in another. We suggested using the first letter of the region of interest (H for Head or N for Neck) in the picture file names and this is where it becomes useful. It allows us to sort all pictures automatically. This is especially useful when dealing with a lot of pictures. When choosing a letter to group picture categories, be cautious of not using a letter or character that appears elsewhere in the filename.

Note: This step is performed via the *Windows command processor* (also known as *command prompt*). An equivalent method should work under other operating systems such as Mac OS or Linux.

Open the command prompt by pressing the windows key (WIN) + R. Then, write *cmd* in the new window.

Be careful: the command prompt is sensitive to caps lock and spaces.



Now, specify the path of the directory where all the pictures are currently saved by using *cd* followed by a space and the directory path. Press enter.

cd Your_directory_path

Then, to create new folders where you will send your pictures (*in our case*, we needed two new folders: one for the head pictures and one of the neck pictures), enter *md* followed by a space and the new folder's name. Separate each folder by a space and press enter.

md Neck Head

Finally, use the *move* command, followed by a space and the recurring pattern specified (*in our case*, *_H_* or *_N_*) between two asterisks. Add a space and specify the name of the destination folder followed by "/" and press enter.

move *_N_* Neck/
move *_H_* Head/

```
C:\Users\Utilisateur>cd C:\Users\Utilisateur\Documents\Coloration

C:\Users\Utilisateur\Documents\Coloration>md Neck Head

C:\Users\Utilisateur\Documents\Coloration> move *_N_* Neck/

C:\Users\Utilisateur\Documents\Coloration>move *_H_* Head/
```

The pictures will now be sorted in their respective new folders according to the region of interest that you want to analyze.

Standardizing pictures before binarization

In most cases, you will need to create a new picture which is standardized. This step is required to remove absolute variation in pixel values (i.e., variations in brightness). For more information on this step see Teasdale et al. [8].

For each picture, the standardization is done on all three color channels by dividing each pixel value by the total value from all color channels (e.g., Standardized red = R/(R+G+B)).

In our case, standardization made it too difficult to distinguish yellow from green during binarization. Thus, we did not standardize our pictures. If you need to standardize your pictures, you can follow the steps below.

Loading R packages

See *Loading R packages* note in *Creating linearization equations* section.

```
library(imageviewer)
library(magick)
library(rsvg)
library(beepr)
library(hexView)
library(tiff)
library(EBImage)
library(rio)
```

Setting the directory

To be able to run these scripts, you need to set your directory to the folder where you saved your linearized, scaled, and cropped pictures in .TIFF format with the function setwd() or by following this path *Session* >> *Set Working Directory* >> *Choose Directory*. You can also use the here package (https://here.r-lib.org/) which is less dependent on the way you organize your files.

Creating a list of pictures

```
in.images <- list.files(path = "images/scaled_pictures", pattern = "_H_", full.
names = F, recursive = F)
in.images
```

```
## [1] "ID_H_linearized_scaled_cut_10.tif"
```

Here, again, you need to confirm that you can access the elements of the list.

```
length(in.images)
```

```
## [1] 1
```

In our case, we have one picture in our list.

Converting pictures to arrays

As in *Linearizing pictures* section, we used the format.image() function to format the pictures correctly (i.e., import, convert to an array, and get values from 0 to 1) and list them.

```
format.image <- function(y) {
  list_array <- list()
  for (n in 1:length(y)) {
    image <- paste("standardized_", y[n], sep = "")
    print(image)
    y2 <- image_read(y[n])
    #print(y2)
    print(image_info(y2))
    y_array <- as.integer(y2[[1]])
    y_array <- transpose(y_array)
    y_array <- y_array / 255
    # print(y_array)
    list_array[[image]] <- y_array
  }
  return(list_array)
}
```

You can apply the function to your list of pictures:

```
picture_list <- format.image(in.images)
```

```
## [1] "standardized_ID_H_linearized_scaled_cut_10.tif"
## # A tibble: 1 × 7
##   format width height colorspace matte filesize density
##   <chr>  <int>  <int> <chr>      <lgl>    <int> <chr>
## 1 TIFF    1381   1381 sRGB       FALSE 11443455 138x138
```

In our case, we can see that we have one picture of 1381 pixels (width) per 1381 pixels (height).

Again, you want to ensure you can access the reflectance values from the first picture. You also need to make sure you select an area of your picture that is not black.

In our case, we indicated that we wanted to see the reflectance values between the pixel rows 670 and 675, and between the pixel columns 662 and 667 of the third color channel (blue).

```
picture_list[[1]][670:675, 662:667, 3]
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
## [1,] 0.2549020 0.2470588 0.2392157 0.2274510 0.2117647 0.1921569
## [2,] 0.2549020 0.2392157 0.2392157 0.2352941 0.2196078 0.2000000
## [3,] 0.2470588 0.2392157 0.2352941 0.2352941 0.2117647 0.2000000
## [4,] 0.2431373 0.2352941 0.2352941 0.2274510 0.2117647 0.2000000
## [5,] 0.2392157 0.2274510 0.2274510 0.2274510 0.2196078 0.2078431
## [6,] 0.2352941 0.2235294 0.2235294 0.2235294 0.2235294 0.2235294
```

In addition, you can verify that the reflectance values are, as supposed, between 0 and 1.

```
summary(picture_list[[1]])
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.00000 0.00000 0.00000 0.03371 0.00000 0.76471
```

Standardizing pictures

We created a function that standardizes all three color channels.

```
standardize.image <- function(x, output_path) {

    for(m in 1:length(x)){
    # name red layer
    r <- x[[m]][, , 1]
    # name green layer
    g <- x[[m]][, , 2]
    # name blue layer
    b <- x[[m]][, , 3]
    # Standardize red channel
    pr <- r / (r + g + b)
    # Standardize green channel
    pg <- g / (r + g + b)
    # Standardize blue channel
    pb <- b / (r + g + b)
    # Dividing by 0 outputs NaN in R. We need to replace NaN with 0 (so that th
ey are equal to the darkest pixel value).
    pr[is.na(pr)] <- 0
    pg[is.na(pg)] <- 0
    pb[is.na(pb)] <- 0
    # stack and convert to image. Order is important in the three layers
    # this image is now standardized (colors might seem weird because of the st
andardization)
    z <- Image(array(dim = c(nrow(pr), ncol(pr), 3), data = cbind(pr, pg, pb)))
    # set to color mode
    colorMode(z) <- Color
    # the new picture will use the origin name, but will add "_standardized" at
the end and will be saved in the .TIFF format
    output_filename <- paste(names(x[m]), "_standardized", ".TIFF", sep = "")
    # specify the path where the standardized pictures will be saved
    output_fullpath <- file.path(output_path, output_filename)
    # save as new image (with "_standardize" added at the end) in 16-bit TIFF f
ormat
    writeImage(z, output_fullpath)
  }
}
```

```
output_folder <- file.path("X:", "Your", "file", "Path", fsep="/")

standardized.image(picture_list, output_folder)
```

The standardized version of our head picture looks like this:

Binarizing pictures

The binarization is used to isolate the color of interest (yellow in this case because we are focusing on head pictures in this example) from all other colors present in our picture. It will give a score of 0 (black) to the color of interest and a score of 1 (white) to the rest. Then, from the binarized pictures, it will compute the number of yellow pixels (black) and the number of pixels that are not considered yellow (white) allowing us to calculate the proportion of yellow pixels in the area of interest. The scripts can be adapted to any color and area of interest. Throughout this section, we will modify a function named binarize.image(), step by step, mainly to show you how to determine the color threshold that will allow you to binarize your pictures, until we arrive at its final form that will allow you to compute the proportion of pixels from your pictures corresponding to your color of interest.

Loading R packages

See *Loading R packages* note in *Creating linearization equations* section.

```
library(imageviewer)
library(magick)
library(rsvg)
library(beepr)
library(hexView)
library(tiff)
library(EBImage)
library(rio)
```

Setting the directory

To be able to run these scripts, you need to set your directory by choosing the folder where you saved your linearized, scaled, and cropped (and possibly standardized) pictures in .TIFF format with the function setwd() or by following this path *Session >> Set Working Directory >> Choose Directory*. You can also use the here package (https://here.r-lib.org/) which is less dependent on the way you organize your files.

Creating a list of pictures

```
in.images <- list.files(path = "images", pattern = "_H_", full.names = F, recur
sive = F)
in.images

## [1] "ID_H_linearized_scaled_cut_10.tif"
```

We can see that our file only included one picture named *ID_H_linearized_scaled_cut_10.tif*.

Converting pictures to arrays

As in *Linearizing pictures* section, we used the format.image() function to format the pictures correctly (i.e., import, convert to an array, and get values from 0 to 1) and list them.

```
format.image <- function(y) {
  list_array <- list()
  for (n in 1:length(y)) {
    image <- paste("10_", y[n], sep = "")
    print(image)
    y2 <- image_read(y[n])
    #print(y2)
    print(image_info(y2))
    y_array <- as.integer(y2[[1]])
    y_array <- transpose(y_array)
    y_array <- y_array / 255
    #print(y_array)
    list_array[[image]] <- y_array
  }
  return(list_array)
}
```

You can apply the function to your list of pictures:

```
picture_list <- format.image(in.images)

## [1] "10_ID_H_linearized_scaled_cut_10.tif"
## # A tibble: 1 × 7
##   format width height colorspace matte filesize density
##   <chr>  <int>  <int> <chr>      <lgl>    <int> <chr>
## 1 TIFF    1381   1381 sRGB       FALSE 11443455 138x138
```

You can see that we had one picture of 1381 pixels (width) per 1381 pixels (height).

Again, you need to ensure you can access the reflectance values from the first picture. You also need to make sure you select an area in your picture that is not black.

In our case, we wanted to see the reflectance values contained between the pixel rows 670 and 675, and between the pixel columns 662 and 667 on the third color channel (blue).

```
picture_list[[1]][670:675, 662:667, 3]
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
## [1,] 0.2549020 0.2470588 0.2392157 0.2274510 0.2117647 0.1921569
## [2,] 0.2549020 0.2392157 0.2392157 0.2352941 0.2196078 0.2000000
## [3,] 0.2470588 0.2392157 0.2352941 0.2352941 0.2117647 0.2000000
## [4,] 0.2431373 0.2352941 0.2352941 0.2274510 0.2117647 0.2000000
## [5,] 0.2392157 0.2274510 0.2274510 0.2274510 0.2196078 0.2078431
## [6,] 0.2352941 0.2235294 0.2235294 0.2235294 0.2235294 0.2235294
```

In addition, you can verify that the reflectance values are, as supposed, between 0 and 1.

```
summary(picture_list[[1]])
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.00000 0.00000 0.00000 0.03371 0.00000 0.76471
```

Determining the color threshold & binarizing the pictures

We created a function that binarizes pictures according to the color of interest. The function creates a new picture where all pixels of our color of interest are black, and all other colors are white.

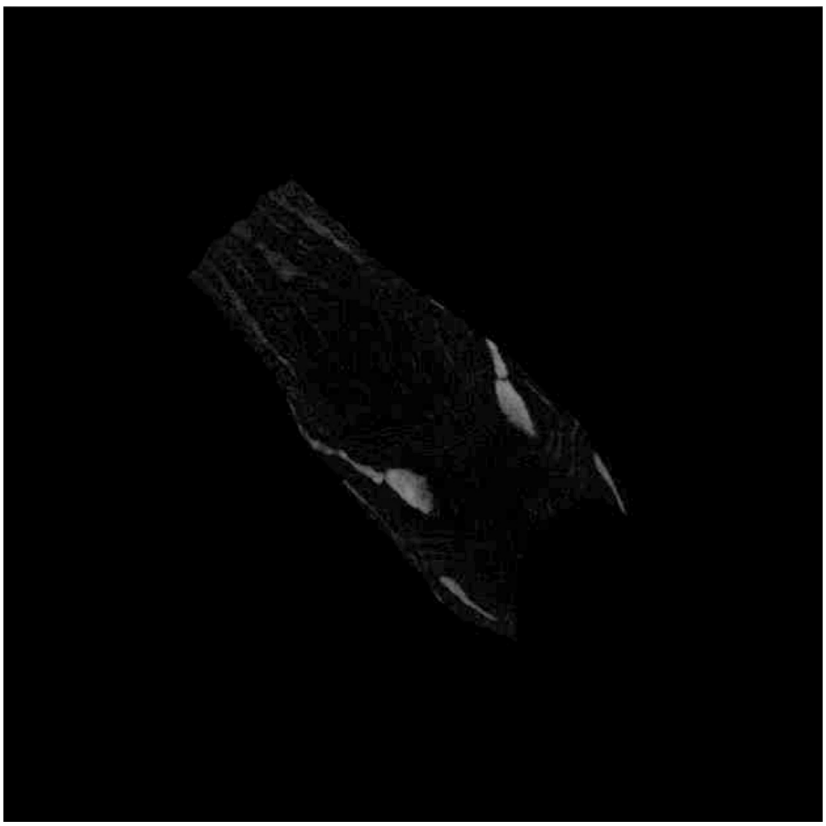
In our case, we wanted to compute the number of yellow pixels relative to the total number of pixels. We first needed to determine which reflectance value corresponded to what we call *yellow* in our pictures (i.e., what our eyes see as yellow). We identified a threshold of reflectance value that is included in our binarization function to determine which pixels are yellow. With this threshold, the function binarizes our pictures according to our color of interest: the color of interest (i.e., yellow) will have a score of 0 (black) and the rest will have a score of 1 (white).

To determine the threshold, we used the first part of the final function that isolates the yellow component of the pictures by substracting the blue channel from the green channel. The formula to use to isolate the color of interest will depend on the color you want to isolate. For each picture, a new picture is saved in a new subfolder (e.g., subfolder named *binarized_pictures*) by adding *yellow_g-b* at the end of the picture name. You can modify the new name as you wish.

In our case, we performed this first step on 50 pictures to select the optimal threshold to binarize our pictures according to the color of interest.

```
binarize.image <- function(x, output_path) {
  # Creates an empty database to compile proportion of colored pixels at the en
d
  Prop_list_yellow_head <- NA
  for(m in 1:length(x)){
    # name red channel
    pr <- x[[m]][, , 1]
    # name green channel
    pg <- x[[m]][, , 2]
    # name blue channel
    pb <- x[[m]][, , 3]
    ## To isolate the yellow component: this part is specific to the color you
want to isolate
    po_yellow <- pg - pb # To isolate yellow, we need to subtract the blue chan
nel from the green channel
    # stack and convert to image. Order is important. This is to create a new p
icture with the yellow component isolated.
    z <- Image(array(dim = c(nrow(po_yellow), ncol(po_yellow), 1), data = po_ye
llow))
    # the new picture will use the origin name, but will add "_yellow_g-b" at t
he end and will be saved in the .TIFF format
    output_filename <- paste(names(x[m]), "_yellow_g-b", ".TIFF", sep = "")
    # specify the path where the pictures will be saved
    output_fullpath <- file.path(output_path, output_filename)
    # save as a new picture in .TIFF format with "_yellow_g-b" added at the end
of the name in the folder selected
    writeImage(z, output_fullpath)
  }
}

output_folder <- file.path("X:", "Your", "file", "Path", fsep="/")

binarize.image(picture_list, output_folder)
```

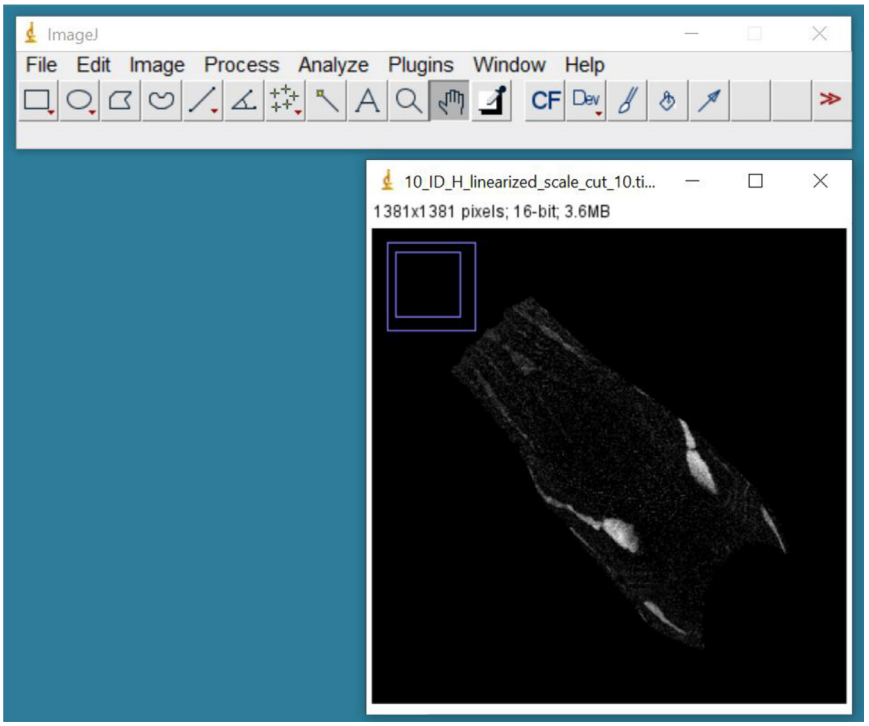
After running this function, you now have a picture that isolates the color of interest, and it should look like this (what you see in white are the yellow areas):



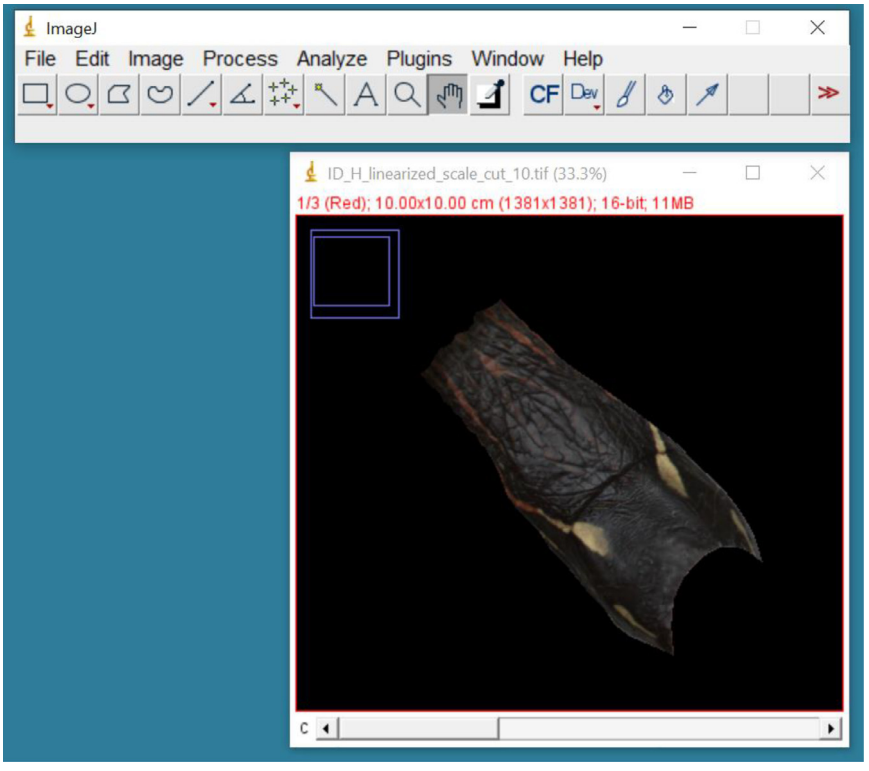
The preview of the pictures may appear darker when viewed with your computer picture viewer compared to ImageJ, it is normal.

A subset of pictures must be used to determine the threshold that will be used to binarize your pictures. This can be done by opening the newly saved pictures with the color of interest isolated in ImageJ and putting your cursor over a region of the color of interest (yellow here) and noting the coordinates. Then, in R, you can find the reflectance value (between 0 and 1) that corresponds to the color of interest. You can also use the version of the picture prior to the isolation of the color of interest given that you will see the real colors and it can be easier to select the coordinates.

Here, you can see our image representing the yellow component isolated from the other colors when we open it in ImageJ:

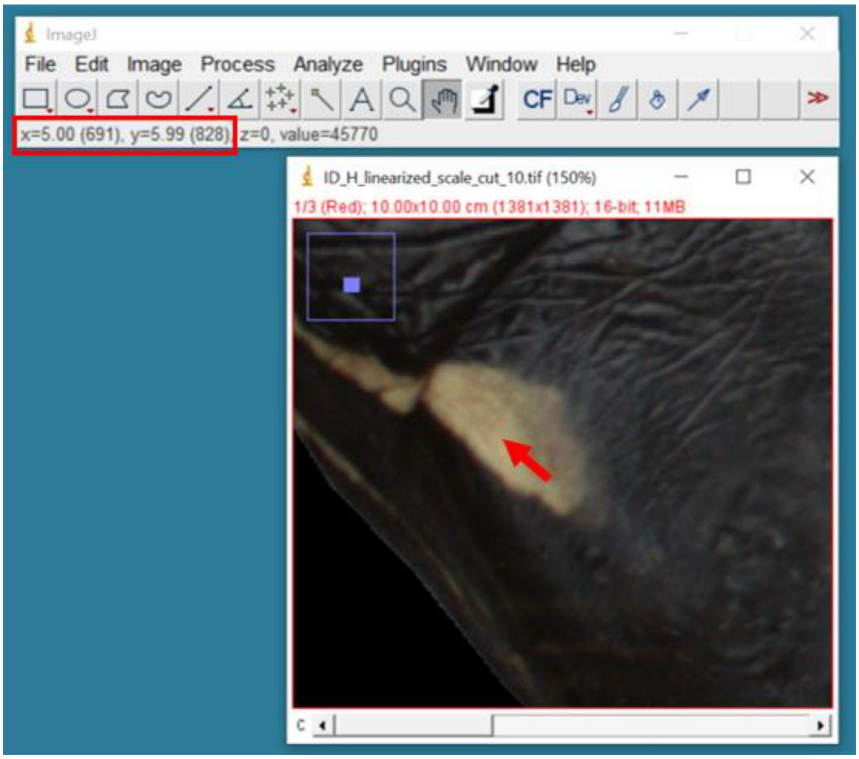


In our case, we opened the picture in ImageJ prior to the isolation of the yellow component to select the coordinates:



ImageJ gives you the coordinates of the pixel where you put your cursor. Move your cursor over a region of your color of interest to get its coordinates.

In our case, we zoomed on a yellow spot of the head and put our cursor over it (red arrow). We then noted the coordinates of the pixel (located in the red box) in a spreadsheet file.



The coordinates of a yellow pixel in this picture were $x = 691$ and $y = 828$.

You will need to do this for all your picture subset (one coordinates per color of interest). After that, you can return in R to determine the reflectance value of these pixels. You can create a new spreadsheet file to i) keep all the coordinates obtained for each picture of the subset and ii) enter the reflectance values that you will obtain afterward in R.

	A	B	C	D	E
1	ID	CoordinateX	CoordinateY	Reflectance-value	
2	ID_H_linearized_cut_10	691	828		
3					
4					
5					
6					

When all the coordinates are identified, you can return in R to a modified version of the previous function that will allow you to determine the reflectance values of the selected pixels coordinates.

Be careful: If you apply the function to your previous picture list (instead of applying it only to the picture from which you got the coordinates), it will determine the reflectance value of the coordinates selected for all the pictures in the list. Since yellow areas of different pictures are not all at the same coordinates, this will lead to incorrect thresholds.

In this example, only one picture was included in our picture list, so we do not have to worry about this.

Here is the modified function, make sure to change the coordinates according to what you obtained in ImageJ:

```
binarize.image <- function(x, output_path) {
  Prop_list_yellow_head <- NA
  for(m in 1:length(x)){
    pr <- x[[m]][, , 1]
    pg <- x[[m]][, , 2]
    pb <- x[[m]][, , 3]
    po_yellow <- pg - pb
    z <- Image(array(dim = c(nrow(po_yellow), ncol(po_yellow), 1), data = po_ye
llow))
    output_filename <- paste(names(x[m]), "_yellow_g-b", ".TIFF", sep = "")
    output_fullpath <- file.path(output_path, output_filename)
    writeImage(z, output_fullpath)
    ## this is the section that you need to modify according to the coordinates
obtained in ImageJ to determine the reflectance value of this selected pi xel.
In our case x = 691 and y = 828
    print(c(z[691, 828, 1], paste("Reflectance values - Yellow pixel")))
    }
  return(Prop_list_yellow_head)
}
```

Now, you can run the new version of the function to have the reflectance value of the coordinates selected:

```
output_folder <- file.path("X:", "Your", "file", "Path", fsep = "/")

Reflectance_value_yellow <- binarize.image(picture_list, output_folder)

## [1] "0.133333333333333"                "Reflectance values - Yellow pixel"
```

You can see that the pixel selected (coordinates in ImageJ: $x = 691$ and $y = 828$) has a reflectance value of **0.13**.

The function can now be modified to binarize (black/white) the pictures according to this reflectance threshold. For now, we will only show you how the function works and how a binarized picture looks based on the reflectance value that we just computed.

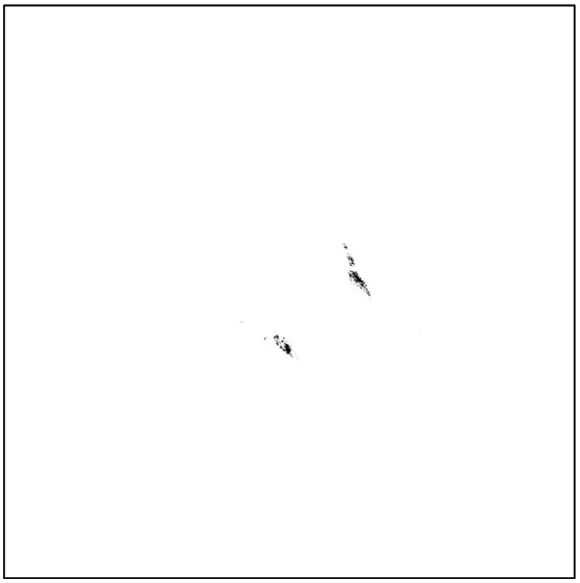
```
binarize.image <- function(x, output_path) {
  Prop_list_yellow_head <- NA
  list_black_head <- NA
  list_white_head <- NA
  for(m in 1:length(x)){
    pr <- x[[m]][, , 1]
    pg <- x[[m]][, , 2]
    pb <- x[[m]][, , 3]
    po_yellow <- pg - pb
    z <- Image(array(dim = c(nrow(po_yellow), ncol(po_yellow), 1), data = po_ye
llow))
    # output_filename <- paste(names(x[m]), "_yellow_g-b", ".TIFF", sep = "")
    # output_fullpath <- file.path(output_path, output_filename)
    # writeImage(z, output_fullpath)
    ## You need to modify this function according to the reflectance value obta
ined. In this example, our threshold is 0.13. Let us select only regions > 0.13
for yellow and create a new binarized picture to see the result.
    # All yellow pixels will be black (0) and all other pixels will be white (1
)
    z[which(z[] <= 0.13)] <- 1
    z[which(z[] > 0.13 & z[] != 1)] <- 0
    # Number of pixels that are not considered yellow
    White_yellow <- print(length(z[which(z[] == 1)]))
    Black_yellow <- print(length(z[which(z[] == 0)])) # Number of yellow pixels
    Prop_yellow <- Black_yellow / (White_yellow + Black_yellow) # Proportion of
yellow pixels
    print(c(Prop_yellow, paste("Proportion of yellow pixels")))
    # The new picture will be saved in .TIFF with the "yellow_0.13_g-b" at the
end of the name
    output_filename <- paste(names(x[m]), "_yellow_0.13_g-b", ".TIFF", sep = ""
)
    output_fullpath <- file.path(output_path, output_filename)
    writeImage(z, output_fullpath)
    # Add the proportion of yellow pixels (and the count of black and white pix
els) to the empty table created at the beginning of the function
    Prop_list_yellow_head[[m]] <- Prop_yellow
    list_black_head[[m]] <- Black_yellow
    list_white_head[[m]] <- White_yellow
    }
  return_list <- list(Prop_list_yellow_head, list_black_head, list_white_head)
  return(return_list)
}
```

```
output_folder <- file.path("X:", "Your", "file", "Path", fsep = "/")

Proportion_list_head <- binarize.image(picture_list)

## [1] 1905941
## [1] 1220
## [1] "0.000639694289050584"        "Proportion of yellow pixels"
```

The output gives you the following information:

- The first row is the number of pixels that are not considered yellow according to the threshold selected (i.e., number of white pixels in the binarized picture).
- The second row is the number of yellow pixels according to the threshold selected (i.e., number of black pixels in the binarized picture).
- The third row is the proportion of yellow on the head.

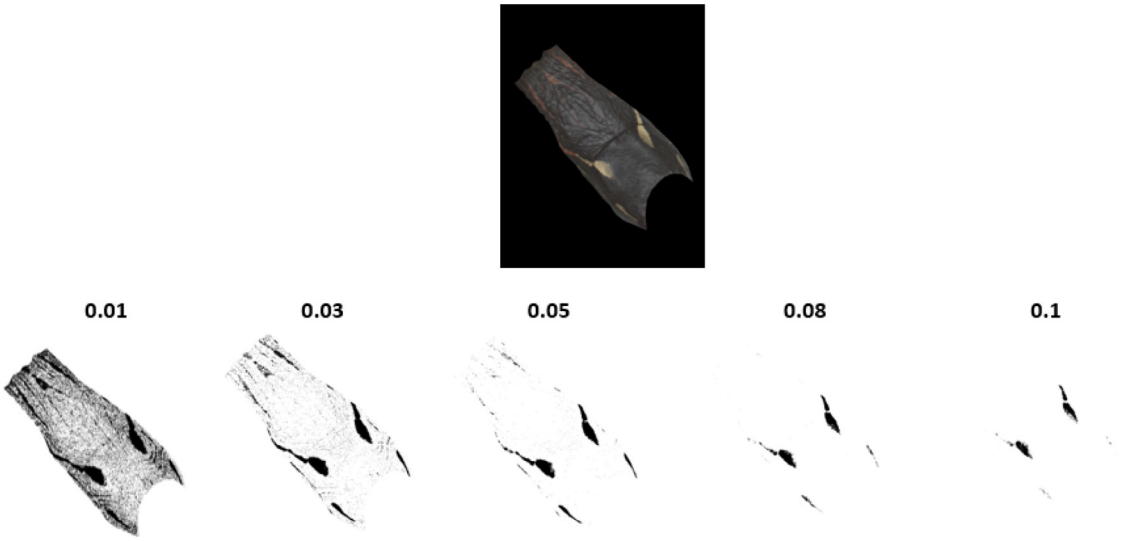
Here is the picture binarized with this threshold:



You can notice that the threshold selected here (0.13) is too restrictive (i.e., we are losing areas of the color of interest (represented in black) according to the original picture). In other words, there should be more black pixels in the binarized picture since the area covered by yellow in the original picture is bigger than what we see in the binarized picture.

By running the previous function with several pictures, you will find an optimal value allowing you to keep only the regions of interest, without losing too much of them. Some trial and error may be required. The black areas in the binarized pictures should only be present where your color of interest is. You can try different thresholds on several pictures and select the one that keeps most of the areas of your color of interest without including too much noise (i.e., areas that do not appear similar visually to your color of interest).

In the following example, the first picture has a threshold of 0.01. This leads to most of the picture being black (not only the yellow patches), which is not restrictive enough. On the other hand, you do not want your threshold to be too restrictive, which would lead to losing your color of interest (i.e., yellow patches (or parts of them) would be white while they should be black).



In our case, we used 50 pictures to determine visually the optimal threshold value to use. We selected a threshold of 0.08 for our shades of yellow.

When the optimal threshold is selected, you can save the proportion of yellow pixels calculated for all your pictures in a database and export it to a spreadsheet file. We modified the list generated by binarize.image() function by adding a new column with the picture name alongside the proportion of yellow calculated. The number of black and white pixels used to calculate the proportion of yellow pixels is also available in the data frame.

```
Proportion_list_head <- as.data.frame(Proportion_list_head)

Proportion_list_head["picture"] <- names(picture_list)

colnames(Proportion_list_head)[1] = "Proportion of yellow pixels (black/[black+
white])"
colnames(Proportion_list_head)[2] = "Number of black pixels"
colnames(Proportion_list_head)[3] = "Number of white pixels"

Proportion_list_head

##   Proportion of yellow pixels (black/[black+white]) Number of black pixels
## 1                                      0.0006396943                    1220
##   Number of white pixels                                  picture
## 1                1905941 10_ID_H_linearized_scaled_cut_10.tif

export(Proportion_list_head, "Head_Picture.xlsx")
```

You should now have your coloration data extracted and ready for further analyses.

Ethics statements

All protocols were approved by animal care committees at the University of Ottawa (protocol BL-3008) and Queen's University (protocol 2018–1836). All fieldwork was conducted under a Parks Canada Agency research and collection permit (number RIC-2018–29178) and Wildlife Scientific Collector's Authorizations from the Ontario Ministry of Natural Resources (numbers 1089358, 1092637 and 1095459). The experiments complied with the ARRIVE guidelines.

Supplementary material and/or additional information [Optional]

Houle-et-al_Reflectance_Values.xlsx: spreadsheet file containing the reflectance values that were extracted in ImageJ from the six grey scales of the color chart. This spreadsheet was filed following the *Extracting reflectance values with ImageJ* section and can be used to practice the R scripts of the *Creating linearization equations* section.
DSC_0022 and DSC_0023: Pictures (.TIFF and .NEF) that were used to create this tutorial and can be used to follow and try this tutorial.
Picture_Section5.tiff: Picture already linearized and cropped by following the *Linearizing pictures* section of this tutorial and can be used to follow and try the *Rescaling pictures and selecting the area of interest* section of this tutorial.
ID_H_linearized_scaled_cut_10.tiff: DSC_0023 that was already linearized, scaled and cropped by following this tutorial. This picture can be used to follow and try the *Standardizing pictures before binarization* and *Binarizing pictures* sections of this tutorial.
Houle-et-al_Tutorial_RMarkDown_Rcodes.html: R Markdown document in html that only contains the sections of this tutorial using R codes: *Creation linearization equations, Linearizing pictures, Standardizing picture before binarization, Binarizing pictures*.
Houle-et-al_Tutorial_Rcodes_Simplified.rmd: R Markdown file containing all the R codes from the sections of this tutorial using R: *Creation linearization equations, Linearizing pictures, Standardizing picture before binarization, Binarizing pictures*. This document is a simplified version of *Houle-et-al_Tutorial_RMarkDown_Rcodes.html*.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Carolyne Houle: Conceptualization, Methodology, Software, Validation, Data curation, Writing – original draft. **Audrey Turcotte:** Conceptualization, Methodology, Validation, Investigation, Writing – original draft. **James E. Paterson:** Methodology, Writing – review & editing. **Gabriel Blouin-Demers:** Conceptualization, Supervision, Writing – review & editing, Funding acquisition. **Dany Garant:** Conceptualization, Supervision, Writing – review & editing, Funding acquisition.

Data availability

All files necessary to run our scripts (e.g., Excel file, pictures, R codes) are available in supplementary material

Acknowledgments

We thank all field assistants who helped with data collection. This work was supported by a Natural Sciences and Engineering Research Council of Canada (NSERC) Strategic Project Grant focused on the Rideau Canal Waterway and by funding from the Ottawa Field-Naturalists' Club. AT was supported by a postgraduate NSERC scholarship.

Supplementary materials

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.mex.2024.102648.

References

[1] I.C. Cuthill, W.L. Allen, K. Arbuckle, B. Caspers, G. Chaplin, M.E. Hauber, G.E. Hill, N.G. Jablonski, C.D. Jiggins, A. Kelber, J. Mappes, J. Marshall, R. Merrill, D. Osorio, R. Prum, N.W. Roberts, A. Roulin, H.M. Rowland, T.N. Sherratt, J. Skelhorn, M.P. Speed, M. Stevens, M.C. Stoddard, D. Stuart-Fox, L. Talas, E. Tibbetts, T. Caro, The biology of color, Science 357 (2017) eaan0221, doi:10.1126/science.aan0221.
[2] M.D. Abràmoff, P.J. Magalhães, S.J. Ram, Image processing with ImageJ, Biophotonics Int. 11 (2004) 36–42 https://imagej.nih.gov/ij/download.html.
[3] D. Coffin, IJ: Plugins: ij-dcraw – Reader for digital camera raw image. (2015) https://ij-plugins.sourceforge.net/plugins/dcraw/.
[4] R. Core Team, R: a language and environment for statistical computing R Foundation for Statistical Computing. (2022) https://www.R-project.org/
[5] Posit TeamRStudio: Integrated Development Environment for R. Posit Software, PBC, Boston, MA, 2022 http://www.posit.co/.
[6] M. Stevens, C.A. Párraga, I.C. Cuthill, J.C. Partridge, T.S. Troscianko, Using digital photography to study animal coloration, Biol. J. Linn. Soc. 90 (2007) 211–237.
[7] J.E. Paterson, G. Blouin-Demers, Distinguishing discrete polymorphism from continuous variation in throat colour of tree lizards, *Urosaurus ornatus*, Biol. J. Linn. Soc. 121 (2017) 72–81, doi:10.1093/biolinnean/blw024.
[8] L.C. Teasdale, M. Stevens, D. Stuart-Fox, Discrete colour polymorphism in the tawny dragon lizard (*Ctenophorus decresii*) and differences in signal conspicuousness among morphs, J. Evol. Biol. 26 (2013) 1035–1046, doi:10.1111/jeb.12115.
[9] X-Rite IncorporatedData, ColorChecker Charts. https://xritephoto.com/documents/literature/en/ColorData-1p_EN.pdf.
[10] J. Troscianko, M. Stevens, Image calibration and analysis toolbox - a free software suite for objectively measuring reflectance, colour and pattern, Methods Ecol. Evol. 6 (2015) 1320–1331, doi:10.1111/2041-210X.12439.