GammaGrid

February 16, 2022

1 Gamma Grid

In this notebook I will calculate a PsychoPy "gammaGrid" from a set of measurements. This is a 4 x 6 matrix, with white (or 'lum') on the first row, and R, G and B guns on the 2nd, 3rd and 4th row. The columns are: mininmum cd/m^2, maximum cd/m^2 and the γ (gamma) parameter in the first three columns. The last 3 columns can stay nan, but they have the a and b for the 0 and 1 type linearization equations used internally by PsychoPy. Type 2 linearization uses linear interpolation between measures, but I haven't discovered how to set measurements, and either way, it is better (faster? easier? more understandable) for a paper to report the γ and that it was corrected for. The last column has a k parameter, which is used by some people, but apparently not by any of the linearization options available in PsychoPy. Maybe it's there for future functionality.

By default the `gammaGrid` looks like this:

[1]:
```
import numpy as np

gammaGrid = np.array([[ 0.,  1.,  1., np.nan, np.nan, np.nan],
                      [ 0.,  1.,  1., np.nan, np.nan, np.nan],
                      [ 0.,  1.,  1., np.nan, np.nan, np.nan],
                      [ 0.,  1.,  1., np.nan, np.nan, np.nan]], dtype=np.
 ↪float32)
```

These values basically say to not change the way the monitor works, particularly the third column where γ is set to 1.

2 Calibration Measurements

And we have some luminance measurements from a monitor in my current lab:

[2]:
```
mycal = np.array(  [[0, 0, 0, 0.09],
                    [17, 17, 17, 1.05],
                    [34, 34, 34, 3.58],
                    [51, 51, 51, 9.16],
                    [68, 68, 68, 15.3],
                    [85, 85, 85, 21.5],
                    [102, 102, 102, 28.1],
                    [119, 119, 119, 34.1],
                    [136, 136, 136, 45.5],
```

```
                    [153, 153, 153, 56],
                    [170, 170, 170, 66.9],
                    [187, 187, 187, 78.9],
                    [204, 204, 204, 92.1],
                    [221, 221, 221, 106.5],
                    [238, 238, 238, 122.3],
                    [255, 255, 255, 136.7],
                    [0, 0, 0, 0.12],
                    [51, 0, 0, 1.47],
                    [102, 0, 0, 4.51],
                    [153, 0, 0, 9.58],
                    [204, 0, 0, 15.8],
                    [255, 0, 0, 25.2],
                    [0, 0, 0, 0.12],
                    [0, 51, 0, 6.55],
                    [0, 102, 0, 19.7],
                    [0, 153, 0, 39.5],
                    [0, 204, 0, 66.5],
                    [0, 255, 0, 98.6],
                    [0, 0, 0, 0.13],
                    [0, 0, 51, 0.39],
                    [0, 0, 102, 1.2],
                    [0, 0, 153, 2.42],
                    [0, 0, 204, 4.26],
                    [0, 0, 255, 7.51]], dtype=np.float32)
```

The first three columns have R, G and B values used as the "stimulus", and the last column has the measured luminance in cd/m^2 when that color was on the monitor.

The first 16 rows should be predicted by the first row of the `gammaGrid`, as it has black-gray-white values. Then there are 6 rows each (only) for the red, green and blue lines in the `gammaGrid` as well.

We will first add a column to the calibration marking each of the guns:

[3]:
```
guns = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,2,2,2,2,2,2,3,3,3,3,3,3]
mycal = np.c_[ mycal, guns ]
```

[4]:
```
(mycal[:,4] == 0).nonzero()[0]
```

[4]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
```

So basically the guns there correspond to the row-index into the `gammaGrid`: - 0 is white - 1 is red - 2 is green - 3 is blue

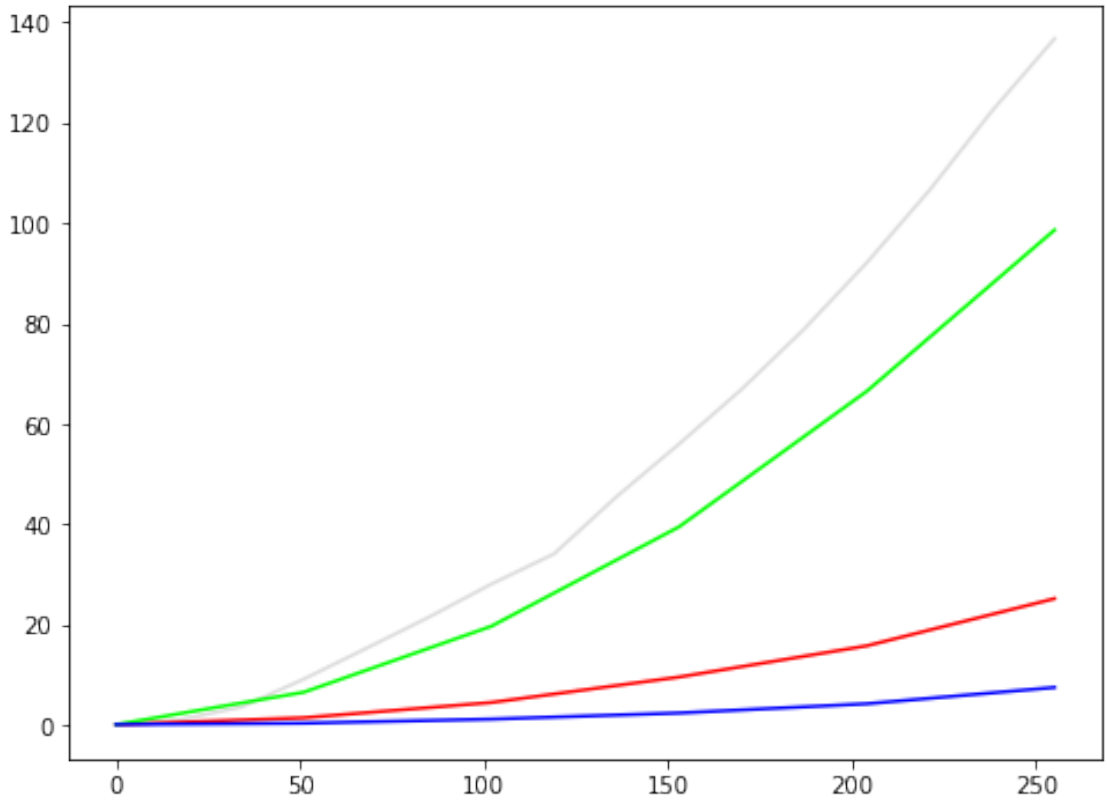
This could be assigned automatically, but I did want to have the "black" measures in all of the guns, and it was faster to just do it manually.

We should also plot this data, to see what it looks like, and we can now do that per gun:

[5]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=[8,6])

for gun in range(4):
    idx = (mycal[:,4] == gun).nonzero()[0]
    if gun == 0:
        rgb = mycal[idx,0]
    else:
        rgb = mycal[idx,gun-1]
    lum = mycal[idx,3]
    col = ['#DDDDDD','#FF0000','#00FF00','#0000FF'][gun]
    plt.plot(rgb, lum, color=col)
```



3 Model Fit

We could estimate the γ parameters for each row of the `gammaGrid` separately, and this would be a decent start. However, the more data is used for estimating the parameters, the better the estimates are. So we could make use of the assumption that the luminance from red, green and blue guns separately, should add up to the luminance from when they are on together.

We'll use `scipy.optimize.minimize` and fit the γ, a and b parameters used in the linearization by PsychoPy. When those parameters predict all our measurements accurately as possible, the values are also the best to use for linearizing the monitor. These are the equations:

type 0: $L = a + (bx)^\gamma$

type 1: $L = (a + bx)^\gamma$

In type 0 linearization, a should be equal to the minimum luminance we measured, but otherwise I'm not sure if it actually matters which version we use (we could test which one predicts the measurements better). Either way, the problem becomes one of minimizing the difference between the measured L on the left-hand side of the equations, and the values predicted by the right-hand side of the equations.

We can start with: - $a = 0$ - $b = 1$ - $\gamma = 2$

As reasonable starting values. Alternatively, setting $b = (max(lum) - a)^{1/\gamma}$ might be more accurate, and adapts to each gun's range of output luminances. Here I use a different estimate that seems to work better for my measurements: $b = (\frac{max(L)}{max(RGB)})^{\gamma^2}$. Either way, the ranges for a and γ should be relatively narrow (a: 0 to maybe 2; γ: between 1 and 3), while b could have many more values depending on the unit of the intensities and which gun it is for.

[6]:
```
def lumModel(par, RGB, eqt=0):
    a, b, g = par

    if eqt == 0:
        return(a + (b*RGB)**g)
    elif eqt == 1:
        return((a + b*RGB)**g)
```

Let's see if this gives somewhat reasonble output:

[7]:
```
idx = (mycal[:,4] == 0).nonzero()[0]
RGB = mycal[idx,0]
par = [0,(136.7/255)**4,2]
print(lumModel(par, RGB))
```

```
[  0.           1.97117197   7.8846879   17.74054777  31.53875159
  49.27929936  70.96219108  96.58742674 126.15500636 159.66492992
 197.11719743 238.51180889 283.8487643  333.12806366 386.34970697
 443.51369422]
```

Seems OK-ish, but particularly for b we will set the bounds to be very wide.

[8]:
```
def lumMSE(pars, mycal, eqt=0):

    errors = []
    pars = np.reshape(pars, [4,3])

    # first we check how well the parameters for each gun match the data:
    for gun in range(4):
```

```
        par = list(pars[gun,:])
        idx = (mycal[:,4] == gun).nonzero()[0]
        RGB = mycal[idx,max(0,gun-1)]

        errors += list( mycal[idx,3] - lumModel(par, RGB, eqt) )

    return(np.mean( np.array(errors)**2 ))


# here are some initial values for parameters, that should be close to optimal:
pars = [0, (136.7/255)**4,  2,
        0,  (25.2/255)**4,  2,
        0,  (98.6/255)**4,  2,
        0,   (7.51/255)**4, 2]

lumMSE(pars, mycal)
```

[8]:
```
9620.299015576053
```

That's still a pretty large error, and that's why we need an optimization function:

[9]:
```
from scipy.optimize import minimize

def lumFit(mycal):

    pars = []
    for gun in range(4):
        idx = (mycal[:,4] == gun).nonzero()[0]
        col = max(0,gun-1)
    pars = [0, (max(mycal[idx,3])/max(mycal[idx,col]))**4,  2,
            0, (max(mycal[idx,3])/max(mycal[idx,col]))**4,  2,
            0, (max(mycal[idx,3])/max(mycal[idx,col]))**4,  2,
            0, (max(mycal[idx,3])/max(mycal[idx,col]))**4,  2]

    bestfit = minimize(lumMSE,
                       pars,
                       args=(mycal),
                       bounds=[(0,2),(0.00001,10000),(1,3),
                               (0,2),(0.00001,10000),(1,3),
                               (0,2),(0.00001,10000),(1,3),
                               (0,2),(0.00001,10000),(1,3)])

    return(bestfit)

fit = lumFit(mycal)
print(fit)
```

```
      fun: 0.37413305827474685
```

```
 hess_inv: <12x12 LbfgsInvHessProduct with dtype=float64>
      jac: array([ 0.00591502,  0.00384571,  0.0061473 , -0.00109627,
-0.01904105,
       -0.00077496, -0.0024543 , -0.01297123,  0.00072768,  0.00728024,
        0.00150243, -0.0147913 ])
  message: b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
     nfev: 4797
      nit: 319
   status: 0
  success: True
        x: array([0.5595004 , 0.06338837, 1.76633488, 0.28866466, 0.02015664,
       1.95937098, 0.45249148, 0.0513495 , 1.78300809, 0.189979  ,
       0.00945845, 2.2341126 ])
```

This should print a dictionary with information about the optimization. Several values are of interest there: - `fun`: the final value of the error function (should be low) - `nfev`: number of function evaluations - `nit`: number of iterations - `success`: hopefully that is True - `x`: the parameter value that resulted in the fit with the lowest error

When I ran this last, `fun` was below 1, and the γ parameters were all reasonable (between 1.7 and 2.3), as were the a parameters (below 1). It's hard to have an intuition for the b parameters, but they seem to scale with the maximum luminance we expect for each of the guns (green is highest, then red, then blue, and the sum of those is close to the value for the "white" gun).

[10]:
```
print(np.reshape(fit['x'], [4,3]))
```

```
[[0.5595004  0.06338837 1.76633488]
 [0.28866466 0.02015664 1.95937098]
 [0.45249148 0.0513495  1.78300809]
 [0.189979   0.00945845 2.2341126 ]]
```

3.1 Added constraint: W = R+G+B

This should work, but it is fitted separately to the 4 guns. I'll now make a fitting function that adds errors for the difference between the white prediction and the sum of the R, G and B predictions.

[11]:
```
def lumMSE(pars, mycal, eqt=0):

    errors = []
    pars = np.reshape(pars, [4,3])

    # first we check how well the parameters for each gun match the data:
    for gun in range(4):
        par = list(pars[gun,:])
        idx = (mycal[:,4] == gun).nonzero()[0]
        RGB = mycal[idx,max(0,gun-1)]

        errors += list( mycal[idx,3] - lumModel(par, RGB, eqt) )
```

```
    # then we also check how well the predictions for individual guns
    # add up to the "white gun" predictions
    # this will determine 1/3 of the errors that need to be minimized
    # so 2/3 of that is actually fitting the data

    RGB = np.linspace(0, 255, np.int32(np.floor(mycal.shape[0]/2)) )
    errors += list( lumModel(pars[0,:], RGB, eqt) - lumModel(pars[1,:], RGB,␣
↪eqt) - lumModel(pars[2,:], RGB, eqt) - lumModel(pars[3,:], RGB, eqt) )

    return(np.mean( np.array(errors)**2 ))
```

So this adds a few errors between predictions, based on the assumption of additivity of the guns. The final error is going to be a bit larger, because 1) there are more constraints, and 2) there are more errors. But the solution might be better, let's see what it is:

[12]:
```
fit = lumFit(mycal)
print(fit)
print(np.reshape(fit['x'], [4,3]))
```

```
      fun: 0.7184265774666075
 hess_inv: <12x12 LbfgsInvHessProduct with dtype=float64>
      jac: array([-0.01128577,  0.12381498,  0.01259927, -0.02048719,
0.00789107,
       -0.00079524, -0.02875353,  0.01117871, -0.00095491, -0.00311517,
        0.01126917, -0.02102649])
  message: b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
     nfev: 2509
      nit: 163
   status: 0
  success: True
        x: array([0.31319442, 0.0639942 , 1.75869716, 0.03050552, 0.02408399,
       1.80429893, 0.05535816, 0.05505831, 1.7430569 , 0.1109068 ,
       0.01348355, 1.74820219])
[[0.31319442 0.0639942  1.75869716]
 [0.03050552 0.02408399 1.80429893]
 [0.05535816 0.05505831 1.7430569 ]
 [0.1109068  0.01348355 1.74820219]]
```

The γ values are now much more similar to eachother, as they should be in the same hardware. The error is indeed larger, but it's still pretty good. I'm going to use this solution.

While I think my measurements are decent, there inevitably is some noise in them, such that the model that I just fitted is probably more accurate. Since the `gammaGrid` also requires the minimum and maximum luminance, and particularly the minimum seemed to be a bit noisier, I'll use the fitted model to see what those values should be:

[13]:
```
for gun in range(4):
    par = np.reshape(fit['x'], [4,3])[gun,:]
    lum = lumModel(par, np.array([0, 255]))
```

```
    print(lum)
```

```
[  0.31319442 136.06271052]
[ 0.03050552 26.47116062]
[5.53581585e-02 1.00036733e+02]
[0.1109068  8.77330798]
```

And here we see the issue with fitting the guns separately: the minimum values are all different (they're also the hardest to measure). The maximum values are relatively OK. But I wonder if it would be better to fit this with a single γ value and a single minimum luminance (a when equation type = 0) and 3 b values, such that the values for the combined gun activity have to equal the sum of the predicted separate gun values, and are not fitted separately.

The problem would be that the values for the b and γ parameters for the "white" or "lum" gun in the `gammaGrid` would have to be determined later.

4 Second model

So let's build a model that fits a single γ and a and four values of b: fewer parameters, more constraints, better fits? This can only work for equation type 0, so that's what we'll do. The RGB input then needs to have R, G and B values, not just one. The fitting fuctions will have be different though

[14]:
```
def LmodelRGB(pars, cal):

    g, a, b0, b1, b2, b3 = pars

    # should be possible to do in 1 line with matrix multiplication:
    R = (b1 * cal[:,0])**g
    G = (b2 * cal[:,1])**g
    B = (b3 * cal[:,2])**g
    L = a + R + G + B

    return(L)

def LmodelW(pars, cal):

    g, a, b0, b1, b2, b3 = pars

    idx = (cal[:,4] == 0).nonzero()[0]
    RGB = cal[idx,0]

    L = a + (b0*RGB)**g

    return(L)
```

Let's see if these functions return more or less OK luminance values:

[15]:
```
pars = [1.8, 0.10, 0.0639, 0.0241, 0.0551, 0.0135]

print(LmodelRGB(pars, mycal))
print(LmodelW(pars, mycal))
```

```
[1.00000000e-01 1.26030545e+00 4.14041827e+00 8.48282871e+00
 1.41695536e+01 2.11241449e+01 2.92907050e+01 3.86255463e+01
 4.90930312e+01 6.06632051e+01 7.33103248e+01 8.70118845e+01
 1.01747939e+02 1.17500614e+02 1.34253744e+02 1.51992594e+02
 1.00000000e-01 1.54962942e+00 5.14790282e+00 1.05731000e+01
 1.76778186e+01 2.63665480e+01 1.00000000e-01 6.52242727e+00
 2.24641907e+01 4.64999437e+01 7.79766353e+01 1.16471117e+02
 1.00000000e-01 6.10772022e-01 1.87861149e+00 3.79016137e+00
 6.29348492e+00 9.35492936e+00]
[1.00000000e-01 1.26067226e+00 4.14169554e+00 8.48547873e+00
 1.41740013e+01 2.11307912e+01 2.92999329e+01 3.86377252e+01
 4.91085191e+01 6.06823506e+01 7.33334684e+01 8.70393595e+01
 1.01780072e+02 1.17537727e+02 1.34296153e+02 1.52040611e+02]
```

Now we need a new error function too, this one does not check additivity:

[16]:
```
def Lerror(pars, mycal):

    idx = (mycal[:,4] == 0).nonzero()[0]
    errorsW = mycal[idx,3] - LmodelW(pars, mycal)

    errorsRGB = mycal[:,3] - LmodelRGB(pars, mycal)

    return(np.mean( np.array(list(errorsW)+list(errorsRGB))**2))
```

And also check if this gives reasonable errors:

[17]:
```
Lerror(pars, mycal)
```

[17]:
```
40.84185214608976
```

Decent, but let's now write an optimization function to bring this down further:

[18]:
```
def Lfit(mycal):

    pars = [1.8, 0.10, 0.0639, 0.0241, 0.0551, 0.0135]

    fit = minimize(Lerror,
                   pars,
                   args=(mycal),
                   bounds=[(1,3),(0,1),(0.00001,10000),(0.00001,10000),(0.
↪00001,10000),(0.00001,10000)])
```

```
    return(fit)

fit = Lfit(mycal)
print(fit)
```

```
      fun: 1.05164376659313
 hess_inv: <6x6 LbfgsInvHessProduct with dtype=float64>
      jac: array([ 4.48818760e-04,  2.83728840e-01,  2.73983058e-03,
6.69406752e-03,
       -5.26689803e-05,  2.55828692e-03])
  message: b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
     nfev: 343
      nit: 45
   status: 0
  success: True
        x: array([1.74739897, 0.        , 0.0654317 , 0.02557761, 0.05480744,
       0.01381505])
```

Interesting! The γ parameter looks like the median of the previous fits, the values for b are very similar to the previous ones, but a is set to 0 now.

4.1 Added constraint: W = R+G+B

We're going to include the additivity of guns in the error function now:

[19]:
```
def Lerror(pars, mycal):

    idx = (mycal[:,4] == 0).nonzero()[0]
    errorsW = mycal[idx,3] - LmodelW(pars, mycal)

    errorsRGB = mycal[:,3] - LmodelRGB(pars, mycal)

    W_RGBerrors = LmodelW(pars, mycal[idx,:]) - LmodelRGB(pars, mycal[idx,:])

    return(np.mean( np.
 ↪array(list(errorsW)+list(errorsRGB)+list(W_RGBerrors))**2))

fit = Lfit(mycal)
print(fit)
```

```
      fun: 0.8127989217006379
 hess_inv: <6x6 LbfgsInvHessProduct with dtype=float64>
      jac: array([ 0.00026577,  0.22817755,  0.00748845, -0.00484531,
-0.00414934,
       -0.00213846])
  message: b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
     nfev: 364
      nit: 47
```

```
  status: 0
 success: True
       x: array([1.74726666, 0.        , 0.06535985, 0.0256306 , 0.05484593,
       0.01389479])
```

Doesn't make much of a difference… good measurements?

Let's see what this fit says for the minimum and maximum luminance values:

[20]:
```
pars = fit['x']

calW = np.array([[0,0,0,0,0],[255,255,255,0,0]])
Wminmax = LmodelW(pars, calW)
print(Wminmax)

calRGB = np.array([[0,0,0,0,0],[255,0,0,0,0],[0,255,0,0,0],[0,0,255,0,0]])
RGBminmax = LmodelRGB(pars, calRGB)
print(RGBminmax)
```

```
[  0.         136.42685243]
[  0.          26.57937128 100.41914265   9.11873091]
```

These values are very similar to the previous ones, except for the zero minimum. The zero minimum suggests that the non-zero measurements in the data might be measurement errors, but who knows.

Now for the gammaGrid:

[21]:
```
gammaGrid
```

[21]:
```
array([[ 0.,  1.,  1., nan, nan, nan],
       [ 0.,  1.,  1., nan, nan, nan],
       [ 0.,  1.,  1., nan, nan, nan],
       [ 0.,  1.,  1., nan, nan, nan]], dtype=float32)
```

Those are still the default values, so we're going to fill in the values we got from the fit:

[22]:
```
gammaGrid[0,1] = Wminmax[1]
gammaGrid[1:,1] = RGBminmax[1:]
gammaGrid[:,2] = fit['x'][0]
gammaGrid
```

[22]:
```
array([[  0.       , 136.42685  ,   1.7472667,         nan,         nan,
                nan],
       [  0.       ,  26.57937  ,   1.7472667,         nan,         nan,
                nan],
       [  0.       , 100.41914  ,   1.7472667,         nan,         nan,
                nan],
       [  0.       ,   9.118731 ,   1.7472667,         nan,         nan,
                nan]], dtype=float32)
```

Now when we have a monitor object, we can use the `.setGammaGrid()` method with the above `gammaGrid` as input. If we then create a window object for this monitor, we can linearize the colors for some awesome psychophysics experiments.

5 Sources

Here are some helpful pages I found in my quest to solve this:

Ingo Fruend's former lab at York University: https://fruendlab.github.io/understanding-gamma-calibration.html This has Python code for a much more straightforward solution to the problem, but it only works for a single gun.

Also for only a single gun, is this page by VPixx's Sophie Kenny: https://vpixx.com/vocal/gammacorrect/ It is meant for Matlab / Psychtoolbox and has an R Shiny app to help you out.