ELECTRIC POTENTIALS AND FIELDS

AND A LITTLE DEEP LEARNING

龚永晖　2015301510059　物基一班

PART 1 ELECTRIC POTENTIALS AND FIELDS

BACKGROUND

An **electric potential** (also called the *electric field potential*, potential drop or the *electrostatic potential*) is the amount of work needed to move a unitpositive charge from a reference point to a specific point inside the field without producing any acceleration. Typically, the reference point is *Earth* or a point at *Infinity*, although any point beyond the influence of the electric field charge can be used.

According to classical electrostatics, electric potential is a scalar quantity denoted by V, equal to the electric potential energy of any charged particle at any location (measured in joules) divided by the charge of that particle (measured in coulombs). By dividing out the charge on the particle a quotient is obtained that is a property of the electric field itself.

This value can be calculated in either a static (time-invariant) or a dynamic (varying with time) electric field at a specific time in units of joules per coulomb ($J\ C^{-1}$), or volts (V). The electric potential at infinity is assumed to be zero.

A generalized electric scalar potential is also used in electrodynamics when time-varying electromagnetic fields are present, but this can not be so simply calculated. The electric potential and the magnetic vector potential together form a four vector, so that the two kinds of potential are mixed under Lorentz transformations.

INTRODUCTION

Classical mechanics explores concepts such as force, energy, potential etc. Force and potential energy are directly related. A net force acting on any object will cause it to accelerate. As an object moves in the direction in which the force accelerates it, its potential energy decreases: the gravitational potential energy of a cannonball at the top of a hill is greater than at the base of the hill. As it rolls downhill its potential energy decreases, being translated to motion, inertial (kinetic) energy.

It is possible to define the potential of certain force fields so that the potential energy of an object in that field depends only on the position of the object with respect to the field. Two such force fields are the gravitational field and an electric field (in the absence of time-varying magnetic fields). Such fields must affect objects due to the intrinsic properties of the object (e.g., mass or charge) and the position of the object.

Objects may possess a property known as electric charge and an electric field exerts a force on charged objects. If the charged object has a positive charge the force will be in the direction of the electric field vector at that point while if the charge is negative the force will be in the opposite direction. The magnitude of the force is given by the quantity of the charge multiplied by the magnitude of the electric field vector.

MODEL

As usual we discretize the independent variables, in this case x, y and z. Points in our space are then specified by integers i, j, and k, with x=i*delta x, y=j*delta y, z=k*delta k. Our goal is to determine the potential V(i, j, k) = V(x=i*delta x, y=j*delta y, z=k*delta k) on this lattice of points. The first step in reaching this goal is to rewrite

$$\frac{\partial^2 V}{\partial x^2}+\frac{\partial^2 V}{\partial y^2}+\frac{\partial^2 V}{\partial z^2}=0$$

as a difference equation. We already know how to write a first derivative in finite-difference form. For example, at the point (i, j, k) the derivative with respect to x may be written as

$$\frac{\partial V}{\partial x}\approx\frac{V(i+1,j,k)-V(i,j,k)}{\Delta x}$$

$$V(i,j,k)=\frac{1}{6}[V(i+1,j,k)+V(i-1,j,k)+V(i,j+1,k)+V(i,j-1,k)+V(i,j,k+1)+V(i,j,k-1)]$$

where we have assumed that the step sizes along x, y, and z are all the same (delta x = delta y = delta z). In words, these simply says that the value of the potential at any point is the average of V at all of the neighboring points. The solution for V(i, j, k) is the function that manages to satisfy this condition at all points simultaneously. From the symmetry it is natural to suppose that the potential on these surfaces of the box will vary linearly with position as we move from x=-1 to x=+1. We assume further that the box is infinite in extent along +-z, so V(i, j, k) is independent of k. We thus have only a two-dimensional problem. Our goal is to find the potential function V(i, j) that satisfies

$$V(i,j,k)=\frac{1}{4}[V(i+1,j,k)+V(i-1,j,k)+V(i,j+1,k)+V(i,j-1,k)]$$

Finally, if we wish to obtain the electric field, this can be calculated by differentiating V(i,j). To do this we use the fact that the component of E in the x direction is

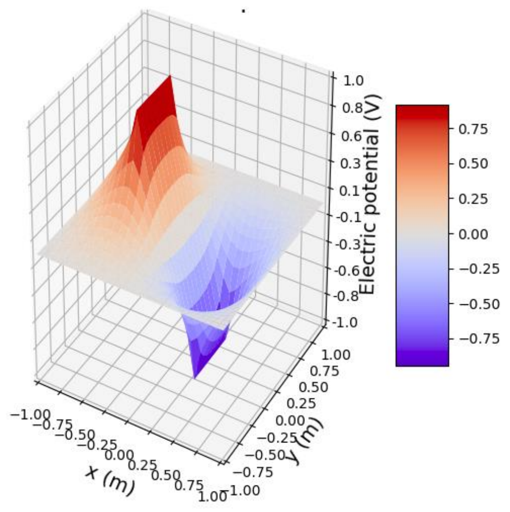
$$E_x = -\frac{\partial V}{\partial x}$$

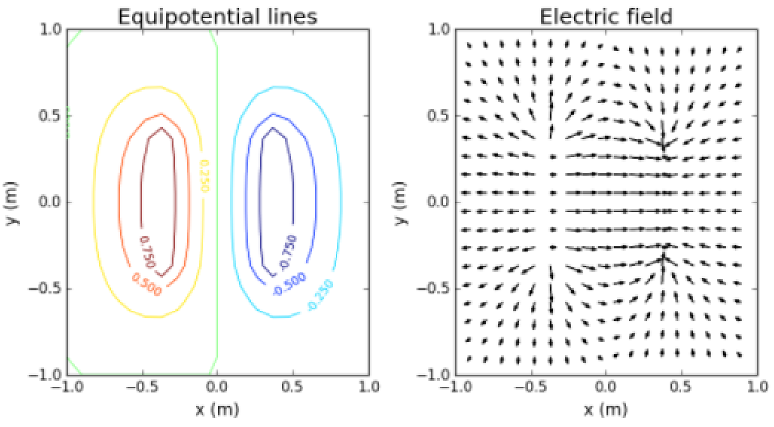
with corresponding relations for E_y and E_x. These dericatives can be estimated using our usual finite difference expressions. In this case we can use a symmetric form for this derivative

$$E_x(i,j) = -\frac{V(i+1,j,k) - V(i-1,j,k)}{2\Delta x}$$

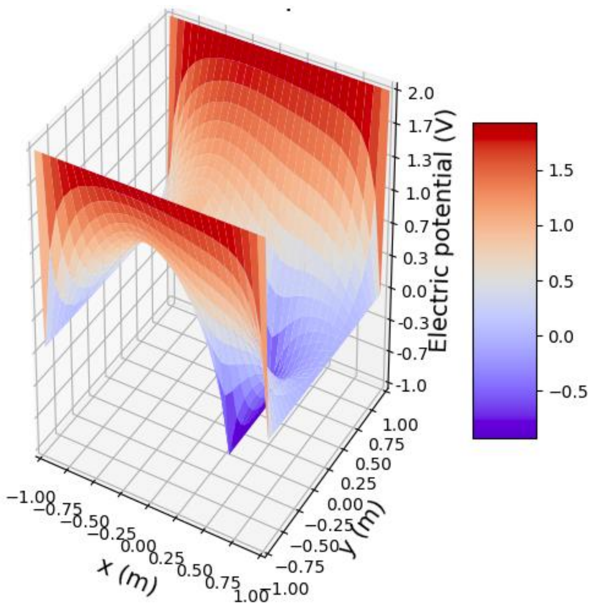
but a lesss symmetric form would give essentially the same results. One note of caution is that E at the boundary needs to be calculated using a one-sided difference equation since, clearly, using values of V at sites beyond the boundary makes no sence.

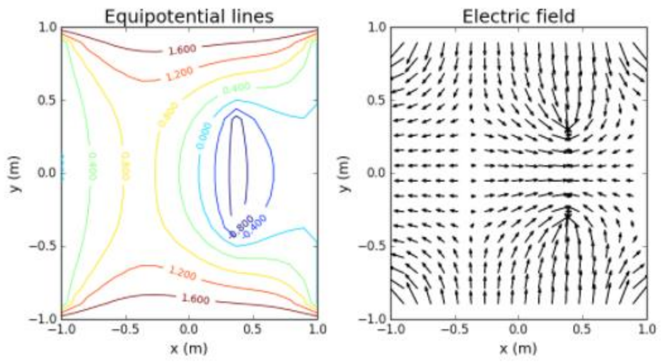
Another interesting use of the relaxation algorithm is the problem of the potential between two parallel capacitor plates. The case of infinite plates can, of course, be handled analytically using Gauss' law. However, here we are intereted in what happens when the plates are finite in extent. Again, we can modify our earlier program to handle this case. All we need to do is set up the proper boundary conditions for V. We set the plates to V=+-1, and the square boundary defined by x=+-1, y=+-1 surrounding the plates is set to V=0. In analytic calculations we would generally apply the condition V=0 at x,y approximate to indinite, but that is usually not pracical in a numerical treatment. Here, for simplicity, we apply this condition on the square boundary just describled; we will have more to say about how to choose such boundary regions and their effects in the next section. After specifying the boundary conditions on V, the application of the relaxation algorithm is the same as that outlined above. The results are givem in





And we will also show in the following with V1=1,V2=-1,V_boundary=2



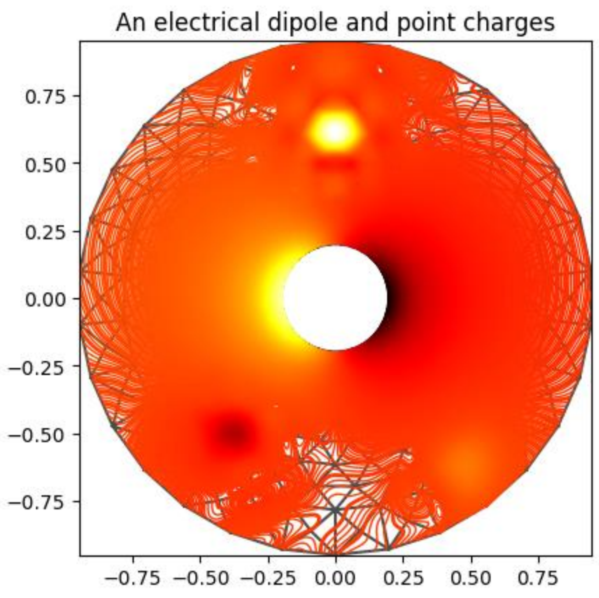


The electric field is seen to be largest between the two plates. In that region the field is approximately uniform (although this is hard to verify with the rather coarse scale used for this plot) and is directed from high to low potential. The fring fields at the edges of and outside the plates are also evident.

AN ELECTRICAL DIPOLE AND POINT CHARGES

```python
from matplotlib.tri import Triangulation, UniformTriRefiner, \
    CubicTriInterpolator
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
import math

pointCharges = []


# -----------------------------------------------------------------------------
# Add points to pointChargesList
# -----------------------------------------------------------------------------
def addPoint(x, y, q):
    global pointCharges
    pointCharges.append([x, y, q])
    return pointCharges


# -----------------------------------------------------------------------------
# Add square 2 grid around the extra point charges
# -----------------------------------------------------------------------------
def addPointGrid(pointX, pointY, x, y):
    xlist = np.linspace(pointX - .1, pointX + .1, num=2)
    ylist = np.linspace(pointY - .1, pointY + .1, num=2)
    xpt, ypt = np.meshgrid(xlist, ylist)
    x2 = np.append(x, xpt)
    y2 = np.append(y, ypt)
    return x2, y2


# -----------------------------------------------------------------------------
# Electrical potential of the pointCharge array
# -----------------------------------------------------------------------------
def pointCharge_potential(x, y):
    V = x * 0
    for k in range(len(pointCharges)):
        x_p = pointCharges[k][0]
        y_p = pointCharges[k][1]
        q_p = pointCharges[k][2]
        r_sq = (x - x_p) ** 2 + (y - y_p) ** 2
        low = 1e-20
        r_sq[r_sq < low] = low
        V += q_p / np.sqrt(r_sq)
    return V


# -----------------------------------------------------------------------------
# Electrical potential of a dipole
# -----------------------------------------------------------------------------
def dipole_potential(x, y):
    """ The electric dipole potential V """
    """ The dipole moment is magnitude 1 aligned in the x axis so that p\dot \hat{R}
    is just cos(theta) """
    r_sq = x ** 2 + y ** 2
    theta = np.arctan2(y, x)
    z = np.cos(theta) / r_sq
    return z


def normalize_potential(V):
    return (np.max(V) - V) / (np.max(V) - np.min(V))


# -----------------------------------------------------------------------------
# Creating a Triangulation
# -----------------------------------------------------------------------------
# First create the x and y coordinates of the points.
n_angles = 30
n_radii = 10
min_radius = 0.2
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2 * math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi / n_angles

x = (radii * np.cos(angles)).flatten()
y = (radii * np.sin(angles)).flatten()

# -----------------------------------------------------------------------------
# Add the point charges by their positions and charges
# -----------------------------------------------------------------------------
pointX = 0
pointY = .617
pointQ = -.01
addPoint(pointX, pointY, pointQ)
```

```python
x, y = addPointGrid(pointX, pointY, x, y)
# print(len(x))

pointX = 0.5
pointY = -.6
pointQ = -.3
addPoint(pointX, pointY, pointQ)
x, y = addPointGrid(pointX, pointY, x, y)
# print(len(x))

pointX = -0.4
pointY = -.5
pointQ = .5
addPoint(pointX, pointY, pointQ)
x, y = addPointGrid(pointX, pointY, x, y)
# print(len(x))



# ----------------------------------------------------------------------------
# Set up the potential from the dipole and the point charges
# ----------------------------------------------------------------------------
V_di = dipole_potential(x, y)
V_mon = pointCharge_potential(x, y)
V = (V_di + V_mon)
# V = V_mon
V = normalize_potential(V)

# Create the Triangulation; no triangles specified so Delaunay triangulation
# created.
triang = Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid * xmid + ymid * ymid < min_radius * min_radius, 1, 0)
triang.set_mask(mask)

# ----------------------------------------------------------------------------
# Refine data - interpolates the electrical potential V
# ----------------------------------------------------------------------------
refiner = UniformTriRefiner(triang)
tri_refi, z_test_refi = refiner.refine_field(V, subdiv=4)

# ----------------------------------------------------------------------------
# Computes the electrical field (Ex, Ey) as gradient of electrical potential
# -----------------------------------------------------------------------------
tci = CubicTriInterpolator(triang, -V)
# Gradient requested here at the mesh nodes but could be anywhere else:
(Ex, Ey) = tci.gradient(triang.x, triang.y)
E_norm = np.sqrt(Ex ** 2 + Ey ** 2)

# -----------------------------------------------------------------------------
# Plot the triangulation, the potential iso-contours and the vector field
# -----------------------------------------------------------------------------
plt.figure()
plt.gca().set_aspect('equal')
plt.triplot(triang, color='0.3')

levels = np.arange(-0., 1., 0.0005)
cmap = cm.get_cmap(name='hot', lut=None)

plt.tricontour(tri_refi, z_test_refi, levels=levels, cmap=cmap,
               linewidths=[2.0, 1.0, 1.0, 1.0])

# Plots direction of the electrical vector field
# plt.quiver(triang.x, triang.y, Ex/E_norm, Ey/E_norm,
#            units='xy', scale=10., zorder=3, color='blue',
#            width=0.007, headwidth=3., headlength=4.)

plt.title('An electrical dipole and point charges')
plt.savefig('DipoleAndPoints.png')
plt.show()
```



PART2 一点点深度学习初步

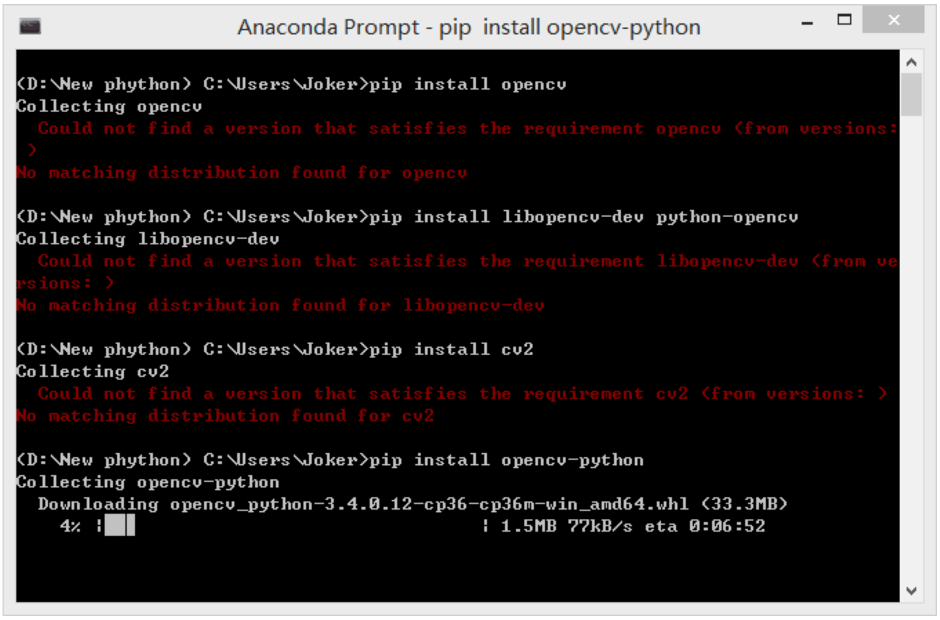
（下面只能用中文写了，因为深度学习英文专业词汇太多，实在看不懂）

深度学习最简单一般从图形处理入手，这次介绍的是目前最流行的处理数据库 Opencv 在深度学习层面的应用。

OpenCV 简介

OpenCV 是计算机视觉领域应用最广泛的开源工具包，基于 C/C++，支持 Linux/Windows/MacOS/Android/iOS ,并提供了 Python ,Matlab 和 Java 等语言的接口，因为其丰富的接口，优秀的性能和商业友好的使用许可，不管是学术界还是业界中都非常受欢迎。OpenCV 最早源于 Intel 公司 1998 年的一个研究项目，当时在 Intel 从事计算机视觉的工程师盖瑞·布拉德斯基(Gary Bradski)访问一些大学和研究组时发现学生之间实现计算机视觉算法用的都是各自实验室里的内部代码或者库 ,这样新来实验室的学生就能基于前人写的基本函数快速上手进行研究。于是 OpenCV 旨在提供一个用于计算机视觉的科研和商业应用的高性能通用库。 第一个 alpha 版本的 OpenCV 于 2000 年的 CVPR 上发布，在接下来的 5 年里，又陆续发布了 5 个 beta 版本，2006 年发布了第一个正式版。2009 年随着盖瑞加入了 Willow Garage，OpenCV 从 Willow Garage 得到了积极的支持，并发布了 1.1 版。2010 年 OpenCV 发布了 2.0 版本，添加了非常完备的 C++接口，从 2.0 开始的版本非常用户非常庞大，至今仍在维护和更新。2015 年 OpenCV 3 正式发布，除了架构的调整，还加入了更多算法，更多性能的优化和更加简洁的 API，另外也加强了对 GPU 的支持，现在已经在许多研究机构和商业公司中应用开来。

安装和使用 OpenCV


Anaconda Prompt - pip install opencv-python

```
(D:\New phython) C:\Users\Joker>pip install opencv
Collecting opencv
  Could not find a version that satisfies the requirement opencv (from versions:
 )
No matching distribution found for opencv

(D:\New phython) C:\Users\Joker>pip install libopencv-dev python-opencv
Collecting libopencv-dev
  Could not find a version that satisfies the requirement libopencv-dev (from ve
rsions: )
No matching distribution found for libopencv-dev

(D:\New phython) C:\Users\Joker>pip install cv2
Collecting cv2
  Could not find a version that satisfies the requirement cv2 (from versions: )
No matching distribution found for cv2

(D:\New phython) C:\Users\Joker>pip install opencv-python
Collecting opencv-python
  Downloading opencv_python-3.4.0.12-cp36-cp36m-win_amd64.whl (33.3MB)
    4% |█                               | 1.5MB 77kB/s eta 0:06:52
```

OpenCV 的结构

和 Python 一样，当前的 OpenCV 也有两个大版本，OpenCV2 和 OpenCV3。相比 OpenCV2，OpenCV3 提供了更强的功能和更多方便的特性。不过考虑到和深度学习框架的兼容性，以及上手安装的难度，这部分先以 2 为主进行介绍。

根据功能和需求的不同，OpenCV 中的函数接口大体可以分为如下部分：

- core：核心模块，主要包含了 OpenCV 中最基本的结构（矩阵，点线和形状等），以及相关的基础运算/操作。

- imgproc：图像处理模块，包含和图像相关的基础功能（滤波，梯度，改变大小等），以及一些衍生的高级功能（图像分割，直方图，形态分析和边缘/直线提取等）。

- highgui：提供了用户界面和文件读取的基本函数，比如图像显示窗口的生成和控制，图像/视频文件的 IO 等。

如果不考虑视频应用，以上三个就是最核心和常用的模块了。针对视频和一些特别的视觉应用，OpenCV 也提供了强劲的支持：

- video：用于视频分析的常用功能，比如光流法（Optical Flow）和目标跟踪等。

- calib3d：三维重建，立体视觉和相机标定等的相关功能。

- features2d：二维特征相关的功能，主要是一些不受专利保护的，商业友好的特征点检测和匹配等功能，比如 ORB 特征。

- object：目标检测模块，包含级联分类和 Latent SVM

- ml：机器学习算法模块，包含一些视觉中最常用的传统机器学习算法。

- flann：最近邻算法库，Fast Library for Approximate Nearest Neighbors，用于在多维空间进行聚类和检索，经常和关键点匹配搭配使用。

- gpu：包含了一些 gpu 加速的接口，底层的加速是 CUDA 实现。

- photo：计算摄像学（Computational Photography）相关的接口，当然这只是个名字，其实只有图像修复和降噪而已。

- stitching：图像拼接模块，有了它可以自己生成全景照片。

- nonfree：受到专利保护的一些算法，其实就是 SIFT 和 SURF。

- contrib：一些实验性质的算法，考虑在未来版本中加入的。

- legacy：字面是遗产，意思就是废弃的一些接口，保留是考虑到向下兼容。

- ocl：利用 OpenCL 并行加速的一些接口。

- superres：超分辨率模块，其实就是 BTV-L1（Biliteral Total Variation – L1 regularization）算法

- viz：基础的 3D 渲染模块，其实底层就是著名的 3D 工具包 VTK（Visualization Toolkit）。

从使用的角度来看，和 OpenCV2 相比，OpenCV3 的主要变化是更多的功能和更细化的模块划分。

基本图像处理

这里我采用一张仙剑图片

进行 Gamma 变换

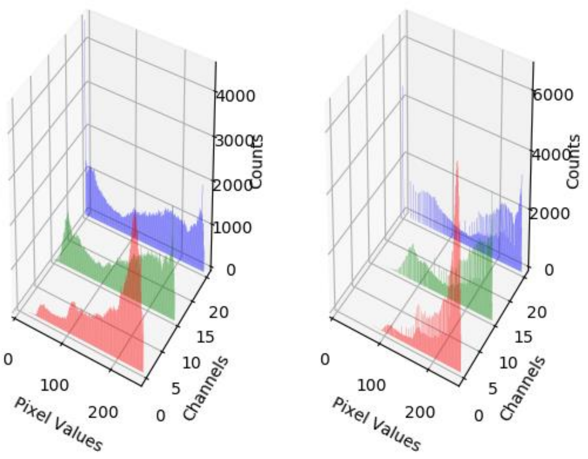
代码

```
import cv2
```

```python
# 读取一张仙剑奇侠传的照片
img = cv2.imread('D:/test1.jpg')

# 缩放成 200x200 的方形图像
img_200x200 = cv2.resize(img, (200, 200))

# 不直接指定缩放后大小，通过 fx 和 fy 指定缩放比例，0.5 则长宽都为原来一半
# 等效于 img_200x300 = cv2.resize(img, (300, 200))，注意指定大小的格式是(宽度,高度)
# 插值方法默认是 cv2.INTER_LINEAR，这里指定为最近邻插值
img_200x300 = cv2.resize(img, (0, 0), fx=0.5, fy=0.5,
                         interpolation=cv2.INTER_NEAREST)

# 在上张图片的基础上，上下各贴 50 像素的黑边，生成 300x300 的图像
img_300x300 = cv2.copyMakeBorder(img, 50, 50, 0, 0,
                                 cv2.BORDER_CONSTANT,
                                 value=(0, 0, 0))

# 对照片中树的部分进行剪裁
patch_tree = img[20:150, -180:-50]

cv2.imwrite('cropped_tree.jpg', patch_tree)
cv2.imwrite('resized_200x200.jpg', img_200x200)
cv2.imwrite('resized_200x300.jpg', img_200x300)
cv2.imwrite('bordered_300x300.jpg', img_300x300)
# 通过 cv2.cvtColor 把图像从 BGR 转换到 HSV
img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

# H 空间中，绿色比黄色的值高一点，所以给每个像素+15，黄色的树叶就会变绿
turn_green_hsv = img_hsv.copy()
turn_green_hsv[:, :, 0] = (turn_green_hsv[:, :, 0]+15) % 180
turn_green_img = cv2.cvtColor(turn_green_hsv, cv2.COLOR_HSV2BGR)
cv2.imwrite('turn_green.jpg', turn_green_img)

# 减小饱和度会让图像损失鲜艳，变得更灰
colorless_hsv = img_hsv.copy()
colorless_hsv[:, :, 1] = 0.5 * colorless_hsv[:, :, 1]
colorless_img = cv2.cvtColor(colorless_hsv, cv2.COLOR_HSV2BGR)
cv2.imwrite('colorless.jpg', colorless_img)

# 减小明度为原来一半
darker_hsv = img_hsv.copy()
darker_hsv[:, :, 2] = 0.5 * darker_hsv[:, :, 2]
darker_img = cv2.cvtColor(darker_hsv, cv2.COLOR_HSV2BGR)
cv2.imwrite('darker.jpg', darker_img)
```

```python
import numpy as np

# 分通道计算每个通道的直方图
hist_b = cv2.calcHist([img], [0], None, [256], [0, 256])
hist_g = cv2.calcHist([img], [1], None, [256], [0, 256])
hist_r = cv2.calcHist([img], [2], None, [256], [0, 256])


# 定义 Gamma 矫正的函数
def gamma_trans(img, gamma):
    # 具体做法是先归一化到 1，然后 gamma 作为指数值求出新的像素值再还原
    gamma_table = [np.power(x / 255.0, gamma) * 255.0 for x in range(256)]
    gamma_table = np.round(np.array(gamma_table)).astype(np.uint8)

    # 实现这个映射用的是 OpenCV 的查表函数
    return cv2.LUT(img, gamma_table)


# 执行 Gamma 矫正，小于 1 的值让暗部细节大量提升，同时亮部细节少量提升
img_corrected = gamma_trans(img, 0.5)
cv2.imwrite('gamma_corrected.jpg', img_corrected)

# 分通道计算 Gamma 矫正后的直方图
hist_b_corrected = cv2.calcHist([img_corrected], [0], None, [256], [0, 256])
hist_g_corrected = cv2.calcHist([img_corrected], [1], None, [256], [0, 256])
hist_r_corrected = cv2.calcHist([img_corrected], [2], None, [256], [0, 256])

# 将直方图进行可视化
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()

pix_hists = [
    [hist_b, hist_g, hist_r],
    [hist_b_corrected, hist_g_corrected, hist_r_corrected]
]

pix_vals = range(256)
for sub_plt, pix_hist in zip([121, 122], pix_hists):
    ax = fig.add_subplot(sub_plt, projection='3d')
    for c, z, channel_hist in zip(['b', 'g', 'r'], [20, 10, 0], pix_hist):
        cs = [c] * 256
        ax.bar(pix_vals, channel_hist, zs=z, zdir='y', color=cs, alpha=0.618,
edgecolor='none', lw=0)

    ax.set_xlabel('Pixel Values')
    ax.set_xlim([0, 256])
    ax.set_ylabel('Channels')
    ax.set_zlabel('Counts')

plt.show()
```

显示结果



此结果对应各通道直方图
左为原图，右为 Gamma 变换后

Gamma 变换是矫正相机直接成像和人眼感受图像差别的一种常用手段，简单来说就是通过非线性变换让图像从对曝光强度的线性响应变得更接近人眼感受到的响应。

REFERENCE

[1] Thanks to https://github.com/luomingyu/computationalphysics_N2013301020045/

[2] Thanks to https://github.com/Cobord/

[3] Thanks to https://en.wikipedia.org/

[4] thanks to https://github.com/frombeijingwithlove/dlcv_for_beginners/