PYTHON IMPLEMENTATION

import math

import random

import collections

import os

import sys

import functools

import operator as op

import numpy as np

import warnings

from scipy.spatial import cKDTree as KDTree

from skimage.filters.rank import entropy

from skimage.morphology import disk, dilation

from skimage.util import img\_as\_ubyte

from skimage.io import imread, imsave

from skimage.color import rgb2gray, rgb2lab, lab2rgb

from skimage.filters import sobel, gaussian\_filter

from skimage.restoration import denoise\_bilateral

from skimage.transform import downscale\_local\_mean

# Returns a random real number in half-open range [0, x).

def rand(x):

r = x

while r == x:

r = random.uniform(0, x)

return r

def poisson\_disc(img, n, k=30):

h, w = img.shape[:2]

nimg = denoise\_bilateral(img, sigma\_range=0.15, sigma\_spatial=15)

img\_gray = rgb2gray(nimg)

img\_lab = rgb2lab(nimg)

entropy\_weight = 2\*\*(entropy(img\_as\_ubyte(img\_gray), disk(15)))

entropy\_weight /= np.amax(entropy\_weight)

entropy\_weight = gaussian\_filter(dilation(entropy\_weight, disk(15)), 5)

color = [sobel(img\_lab[:, :, channel])\*\*2 for channel in range(1, 3)]

edge\_weight = functools.reduce(op.add, color) \*\* (1/2) / 75

edge\_weight = dilation(edge\_weight, disk(5))

weight = (0.3\*entropy\_weight + 0.7\*edge\_weight)

weight /= np.mean(weight)

weight = weight

max\_dist = min(h, w) / 4

avg\_dist = math.sqrt(w \* h / (n \* math.pi \* 0.5) \*\* (1.05))

min\_dist = avg\_dist / 4

dists = np.clip(avg\_dist / weight, min\_dist, max\_dist)

def gen\_rand\_point\_around(point):

radius = random.uniform(dists[point], max\_dist)

angle = rand(2 \* math.pi)

offset = np.array([radius \* math.sin(angle), radius \* math.cos(angle)])

return tuple(point + offset)

def has\_neighbours(point):

point\_dist = dists[point]

distances, idxs = tree.query(point,

len(sample\_points) + 1,

distance\_upper\_bound=max\_dist)

if len(distances) == 0:

return True

for dist, idx in zip(distances, idxs):

if np.isinf(dist):

break

if dist < point\_dist and dist < dists[tuple(tree.data[idx])]:

return True

return False

# Generate first point randomly.

first\_point = (rand(h), rand(w))

to\_process = [first\_point]

sample\_points = [first\_point]

tree = KDTree(sample\_points)

while to\_process:

# Pop a random point.

point = to\_process.pop(random.randrange(len(to\_process)))

for \_ in range(k):

new\_point = gen\_rand\_point\_around(point)

if (0 <= new\_point[0] < h and 0 <= new\_point[1] < w

and not has\_neighbours(new\_point)):

to\_process.append(new\_point)

sample\_points.append(new\_point)

tree = KDTree(sample\_points)

if len(sample\_points) % 1000 == 0:

print("Generated {} points.".format(len(sample\_points)))

print("Generated {} points.".format(len(sample\_points)))

return sample\_points

def sample\_colors(img, sample\_points, n):

h, w = img.shape[:2]

print("Sampling colors...")

tree = KDTree(np.array(sample\_points))

color\_samples = collections.defaultdict(list)

img\_lab = rgb2lab(img)

xx, yy = np.meshgrid(np.arange(h), np.arange(w))

pixel\_coords = np.c\_[xx.ravel(), yy.ravel()]

nearest = tree.query(pixel\_coords)[1]

i = 0

for pixel\_coord in pixel\_coords:

color\_samples[tuple(tree.data[nearest[i]])].append(

img\_lab[tuple(pixel\_coord)])

i += 1

print("Computing color means...")

samples = []

for point, colors in color\_samples.items():

avg\_color = np.sum(colors, axis=0) / len(colors)

samples.append(np.append(point, avg\_color))

if len(samples) > n:

print("Downsampling {} to {} points...".format(len(samples), n))

while len(samples) > n:

tree = KDTree(np.array(samples))

dists, neighbours = tree.query(np.array(samples), 2)

dists = dists[:, 1]

worst\_idx = min(range(len(samples)), key=lambda i: dists[i])

samples[neighbours[worst\_idx][1]] += samples[neighbours[worst\_idx][0]]

samples[neighbours[worst\_idx][1]] /= 2

samples.pop(neighbours[worst\_idx][0])

color\_samples = []

for sample in samples:

color = lab2rgb([[sample[2:]]])[0][0]

color\_samples.append(tuple(sample[:2][::-1]) + tuple(color))

return color\_samples

def render(img, color\_samples):

print("Rendering...")

h, w = [2\*x for x in img.shape[:2]]

xx, yy = np.meshgrid(np.arange(h), np.arange(w))

pixel\_coords = np.c\_[xx.ravel(), yy.ravel()]

colors = np.empty([h, w, 3])

coords = []

for color\_sample in color\_samples:

coord = tuple(x\*2 for x in color\_sample[:2][::-1])

colors[coord] = color\_sample[2:]

coords.append(coord)

tree = KDTree(coords)

idxs = tree.query(pixel\_coords)[1]

data = colors[tuple(tree.data[idxs].astype(int).T)].reshape((w, h, 3))

data = np.transpose(data, (1, 0, 2))

return downscale\_local\_mean(data, (2, 2, 1))

if \_\_name\_\_ == "\_\_main\_\_":

warnings.simplefilter("ignore")

img = imread(sys.argv[1])[:, :, :3]

print("Calibrating...")

mult = 1.02 \* 500 / len(poisson\_disc(img, 500))

for n in (100, 300, 1000, 3000):

print("Sampling {} for size {}.".format(sys.argv[1], n))

sample\_points = poisson\_disc(img, mult \* n)

samples = sample\_colors(img, sample\_points, n)

base = os.path.basename(sys.argv[1])

with open("{}-{}.txt".format(os.path.splitext(base)[0], n), "w") as f:

for sample in samples:

f.write(" ".join("{:.3f}".format(x) for x in sample) + "\n")

imsave("autorenders/{}-{}.png".format(os.path.splitext(base)[0], n),

render(img, samples))

print("Done!")