Computer Graphics (0368-3236) Fall 2019, Excercise I

In this excercise you will implement basic image processing operations, such as scaling and rotation.
You will implement several sampling methods, such as Nearest Neighbour, Bilinear, and Parametric Gaussian.
As you've seen in class, it is preferred to use backward mapping instead of forward mapping, which will gurentee that the target image will be complete. You will display your results, to verify your code.

You will use Python 3 on a Jupyter Notebook environment with scientific packages. There are two ways to get started:

1. Get the **conda** package management and environment system conda from [here (https://docs.conda.io/projects/conda/en/latest/user-guide/install/#regular-installation)](https://docs.conda.io/projects/conda/en/latest/user-guide/install/#regular-installation). Conda is the preferred tool for data and algorithms development in the academia and the industry, but we can not provide support for students who have difficulities with it.
2. Alternatively, you can manually install the needed packages. We will use **Python 3.7** or above, and some very useful packages such as **OpenCV** or **Pillow** for image deserialization, **Imageio** for animation image serialization, **IPython** for notebook support, **Matplotlib** for plotting, **NumPy** for tensor data structures, **SciPy** for statistics functions, and **scikit-image** for image comparison metrics.
   After installing python, run these commands:
   - pip install pip --upgrade
     - or: python -m pip install pip --upgrade
     - or: python3 -m pip install pip --upgrade
     - or: pip3 install pip --upgrade
   - pip install jupyter numpy scipy matplotlib opencv-python pillow scikit-image imageio --upgrade
     - or: python -m pip install jupyter numpy scipy matplotlib opencv-python pillow scikit-image imageio --upgrade
     - or: python3 -m pip install jupyter numpy scipy matplotlib opencv-python pillow scikit-image imageio --upgrade
     - or: pip3 install jupyter numpy scipy matplotlib opencv-python pillow scikit-image imageio --upgrade

Please post your questions on Moodle.

These are some imports, classes and functions implemented for you. You do not need to edit this section. Specifically, **you may not add imports anywhere else in the notebook**.

In [1]:

```
# Copyright (c) 2019 Mattan Serry,
# Computer Graphics and Vision Lab, School of Computer Science, Tel Aviv University
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.

from collections import Counter
from math import sqrt, ceil, floor, radians, cos, sin
from os import remove
from sys import float_info, version_info
from random import randint, uniform
from time import process_time, sleep
from unittest import TestCase
assert version_info >= (3, 7)

# from cv2 import imread, cvtColor, COLOR_BGR2RGB
from IPython.display import HTML, display, Image
from matplotlib import rcParams
from matplotlib.pyplot import show, figure
from numpy import ndarray, iinfo, full, clip, average, array, empty, errstate, uint
8
from PIL import Image as PILImage
from scipy.stats import norm as normal_dist
from skimage import __version__ as skimage_version
assert skimage_version >= '0.16.2'


display(HTML("<style>.container { width:100% !important; }</style>"))


class Timer:
    def __enter__(self):
        self.start = process_time()
        print('*'*16)
        return self

    def __exit__(self, *args):
        self.end = process_time()
        self.interval = self.end - self.start
        print(f'Elapsed: {self.interval:.2f} seconds')
        print('*'*16)


class MyImage:

    def __init__(self, tensor: ndarray):
        self._tensor = tensor
        self._height, self._width, self._channels = tensor.shape
        self._time = None
```

Start by implementing a forward mapping operator. If you scale an image by **sy** and **sy**, where will each pixel translate to?

Part 1: Sampling Methods

You will need to implement the sampling methods *nearest_neighbour*, *bilinear* and *parametric_gaussian*. Make sure the returning value class is *PartialPixels*, which is an extension of *dict*. *PartialPixels* represents the weights of the source pixels per a target pixel, by method.

For example, NN should always return a PartialPixels of size 1, with weight 1.0.

Bilinear should return PartialPixels of size 1, 2 or 4, depends on the value of the source pixel. We relax the demand that the weights are summed to 1.

Parametric Gaussian should return PartialPixels of size up to 9, by calculating over a regular grid of 3x3 with stride **d**. Again we relax the demand that the weights are summed to 1.

We are prociding some clipping and rounding methods for your assistance.

In [2]:
```
def clip_to_range(z: float, *, max_dim: int):
    return clip(z, 0, max_dim - 1)

def clip_and_round(z: float, *, max_dim: int):
    return clip_to_range(z, max_dim=max_dim).round().astype(int)
```

Implement *nearest_neighbour* and validate it. We recommend you use *clip_and_round*.

In [3]:
```
def nearest_neighbour(height: int, width: int, y: float, x: float) -> PartialPixel
s:
    # TODO
    return result

class NNTest(TestCase):
    def test(self):
        height, width = 768, 1024
        y1, x1 = -3.3, 58.12
        y2, x2 = 674.91, 3007.8

        nn1 = nearest_neighbour(height=height, width=width, y=y1, x=x1)
        nn2 = nearest_neighbour(height=height, width=width, y=y2, x=x2)

        print(nn1)
        print(nn2)

        # assert return type is class PartialPixels
        self.assertTrue(isinstance(nn1, PartialPixels))
        self.assertTrue(isinstance(nn2, PartialPixels))

        # assert that NN maps to exactly one pixel (the rounded), and the weight is
as expected
        self.assertTrue({(0, 58): 1.0} == nn1)
        self.assertTrue({(675, 1023): 1.0} == nn2)


NNTest().test()
```

```
{(0, 58): 1.0}
{(675, 1023): 1.0}
```

Implement *bilinear* and validate it. We recommend you to create a *Counter* instance, populate it, and then cast it to *PartialPixels*. We recommend you use *clip_to_range*.

In [4]:
```
def bilinear(height: int, width: int, y: float, x: float) -> PartialPixels:
    # TODO
    return result

class BLTest(TestCase):
    def test(self):
        height, width = 768, 1024
        y1, x1 = 50.0, 60.0
        y2, x2 = 50.25, 60.0
        y3, x3 = 50.25, 60.50

        bl1 = bilinear(height=height, width=width, y=y1, x=x1)
        bl2 = bilinear(height=height, width=width, y=y2, x=x2)
        bl3 = bilinear(height=height, width=width, y=y3, x=x3)

        print(bl1)
        print(bl2)
        print(bl3)

        bl1.normalize()
        bl2.normalize()
        bl3.normalize()

        print(bl1)
        print(bl2)
        print(bl3)


        # assert return type is class PartialPixels
        self.assertTrue(isinstance(bl1, PartialPixels) and isinstance(bl2, PartialP
ixels) and isinstance(bl2, PartialPixels))

        # assert that NN maps to exactly one pixel (the rounded), and the weight is
as expected
        self.assertTrue(
            {
                (50, 60): 1.0,
            } == bl1
        )

        self.assertTrue(
            {
                (50, 60): 0.75,
                (51, 60): 0.25,
            } == bl2
        )


        self.assertTrue(
            {
                (50, 60): 0.375,
                (51, 60): 0.125,
                (50, 61): 0.375,
                (51, 61): 0.125,
            } == bl3
        )


BLTest().test()
```

```
{(50, 60): 1.0}
{(50, 60): 0.75, (51, 60): 0.25}
{(50, 60): 0.375, (51, 60): 0.125, (50, 61): 0.375, (51, 61): 0.125}
{(50, 60): 1.0}
{(50, 60): 0.75, (51, 60): 0.25}
{(50, 60): 0.375, (51, 60): 0.125, (50, 61): 0.375, (51, 61): 0.125}
```

Implement *parametric_gaussian* and validate it. We recommend you to create a *Counter* instance, populate it, and then cast it to *PartialPixels*. We recommend you use *clip_to_range* for grid values, *normalnd* for the sampling value, and *bilinear* to project from fractional pixels to integer pixels.

Important: When you sample with *normalnd*, add **epsilon** to the result. Small values can cause numerical errors.

In [5]:

```python
def parametric_gaussian(height: int, width: int, y: float, x: float, std: float, d: float, epsilon=float_info.epsilon):
    # TODO
    return result


from functools import partial
gaussian03 = partial(parametric_gaussian, std=0.3, d=0.025)
gaussian03.__name__ = 'gaussian 0.3'


class PGTest(TestCase):
    def test(self):
        height, width = 768, 1024
        y1, x1 = 0.001, 10.5

        pg = gaussian03(height=height, width=width, y=y1, x=x1)

        print(pg)

        pg.normalize()

        print(pg)

        # assert return type is class PartialPixels
        self.assertTrue(isinstance(pg, PartialPixels))

        # Verify values of the Gaussian sampling
        self.assertAlmostEqual(pg[(0, 10)], 0.5, places=2)
        self.assertAlmostEqual(pg[(0, 11)], 0.5, places=2)
        self.assertGreater(pg[(0, 11)], pg[(0, 10)])

        self.assertAlmostEqual(pg[(1, 10)], 0.0, places=2)
        self.assertAlmostEqual(pg[(1, 11)], 0.0, places=2)
        self.assertAlmostEqual(pg[(1, 10)], pg[(1, 11)])



PGTest().test()
```

```
{(0, 10): 7.858954823217127, (0, 11): 7.858954823217129, (1, 10): 0.071215723910
53246, (1, 11): 0.07121572391053246}
{(0, 10): 0.4955098239383813, (0, 11): 0.4955098239383814, (1, 10): 0.0044901760
61618693, (1, 11): 0.004490176061618693}
```

In [6]:
```
def calculate_all_partial_pixels_by_method(height: int, width: int, method: callabl
e, backward_map: dict) -> ndarray:
    result = {}

    for (new_y, new_x), (old_y, old_x) in backward_map.items():
        result[(new_y, new_x)] = method(height, width, old_y, old_x)

    return result
```

Part 2: Scale Transformation

We will experiment with a photo of a mandrill. Make sure you can load the image and see the monkey.

In [7]:
```
from base64 import b64decode
from io import BytesIO
```

mandrill = "/9j/4RxRRXhpZgAATU0AKgAAAAgABwESAAMAAAABAAEAAAEaAAUAAAABAAAAYgEbAAUAAAA
BAAAAagEoAAMAAAABAAIAAAExAAIAAAAiAAAAcgEyAAIAAAAUAAAAlIdpAAQAAAABAAAAqAAAANQACvyAAA
AnEAAK/IAAACcQQWRvYmUgUGhvdG9zaG9wIENDIDIwMTUgKFdpbmRvd3MpADIwMTk6MTE6MDggMDA6MjE6M
jEAAAOgAQADAAAAAQABAACgAgAEAAAAAQAAAJugAwAEAAAAAQAAANAAAAAAAAAABgEDAAMAAAABAAYAAAEa
AAUAAAABAAABIgEbAAUAAAABAAABKgEoAAMAAAABAAIAAAIBAAQAAAABAAABMgICAAQAAAABAAAbFwAAAAA
AAABIAAAAAQAAAEgAAAAB/9j/7QAMQWRvYmVfQ00AAf/uAA5BZG9iZQBkgAAAAAH/2wCEAAwICAgJCAwJCQ
wRCwoLERUPDAwPFRgTExUTExgRDAwMDAwMEQwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwBDQsLDQ4NE
A4OEBQODg4UFA4ODg4UEQwMDAwMEREMDAwMDAwRDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDP/AABEI
AKAAdwMBIgACEQEDEQH/3QAEAAj/xAE/AAABBQEBAQEBAQAAAAAAAAADAAECBAUGBwgJCgsBAAEFAQEBAQE
BAAAAAAAAAAEAAgMEBQYHCAkKCxAAAQQBAwIEAgUHBggFAwwzAQACEQMEIRIxBUFRYRMicYEyBhSRobFCIy
QVUsFiMzRygtFDByWSU/Dh8WNzNRaisoMmRJNUZEXCo3Q2F9JV4mXys4TD03Xj80YnlKSFtJXE1OT0pbXF1
eX1VmZ2hpamtsbW5vY3R1dnd4eXp7fH1+f3EQACAgECBAQDBAUGBwcGBTUBAAIRAyExEgRBUWFxIhMFMoGR
FKGxQiPBUtHwMyRi4XKCkkNTFWNzNPElBhaisoMHJjXC0kSTVKMXZEVVNnRl4vKzhMPTdePzRpSkhbSVxNT
k9KW1xdXl9VZmdoaWprbG1ub2JzdHV2d3h5ent8f/2gAMAwEAAhEDEQA/AOS+zlhi15Lgfa0ckj91a9Nm1z
WzILCdToqt1Qpt2Ws3Vu1Y/wAf/JKvSG336FzQ3VpGgiOFU4QiWPone91RFllbts/SEka/99QrrrwQK372O
1mdY/rK9iuurxTU6H1CA555gmNrv5ShlYVLANs1SQS4Tz3bwgIhecdao8NjrXbXzPGp8FedUxtbmyQ0+sQJ
Ht/FRLRQAGuLjETqSf6v8rak2Q4yXAgAbmhs6+Drvbu/ca9OjiiBclDH1Q/Zdd1JLQ36Q8D3UjW5j97vc8C
RE6RrMKVYG/adoLYaJkxP5u1v/VfvpzZ+lcSd4AIBmTHGn8n8+t6RxwkdqKpY92u7Ide9oktfzHkD/moxsc
0kv1AALSD4f+TT2YvqbXshrtA4jiJ92lQyqHPAh20T7h8OyryhKGQR1MZMUQYyru28W7Ez6n42Xy4QxxOoP
5rm/wAtrlUf0xuHWWWfpHAwLPEDX/PWf1C19drW1aQ0EmO60fXvy6a6+X6EE/D81TS+Xt/BklVdiidlCj02
2Uk1uME8/HcrGT9ncxuxpY3byJn8EUmuuposZueNSDqZCpnOt9QmsAtPG7ny/qtUX6O6wfL+1Qe1tZq9R7i
72z7pGnKSmLbTMsa1w4Hy3f8ASSRv/GVxeOr/AP/QqCn7ZjMZkBrLGwAfyrOAbVkNDYLD38D1Tmls9Z7ne
9pkR2j93RQyMJ8+u2WhzgHaaidGuUAiynytrA+viPFA3DfLB4yrXbJ97vc8/vfzHkD/moxsc
0kv1AALSD4f+TT2YvqbXshrtA4jiJ921QyqHPAh20T7h8OyryhKGQR1MZMUQYyru28W7Ez6n42Xy4QxxOoP
5rm/wAtrlUf0xuHWWWfpHAwLPEDX/PWf1C19drW1aQ0EmO60fXvy6a6+X6EE/D81TS+Xt/BklVdiidlCj02
2Uk1uME8/HcrGT9ncxuxpY3byJn8EUmuuposZueNSDqZCpnOt9QmsAtPG7ny/qtUX6O6wfL+1Qe1tZq9R7i
72z7pGnKSmLbTMsa1w4Hy3f8ASSRv/GVxeOr/AP/QqCn7ZjMZkBrLGwAfyrOAbVkNDYLD38D31Xmts9Z7ne
9pkR2j93RQyMJ8+u2WhzgHaaidGuUAiynytrA+viPFA3DfLB4wfaFodLaHUmyyJeJYTPb2ln735qj0/pjqL
JMEkSdvE/ym/nbf8Wm1fc8kJ8UxFrGXEUAYRpWrTrC+deTgSyuvl1nCvwdaYNTG3ePU2HZaRt5YBGvuzHbE
wfd7Xyx30of+dqfWG218OnseTBZS1vGRGvdnzNTxJAmyX6jDZXF/tQHb+YAjLt+4pHv1AtJ/Q2wUk65MP4QdbTkTn
lPYyRaoVNqZe0IKeY7ZPU7TEY0Yf8T0d9BuFd8fNG0oBy9f4UFq2hi5afCCdgDtVDA4GtDqzU+A09qjbYwE
ABNZeIaWXXKlM+aFXJfd6hP3DxHZ9BXUBcLnuYNENG9cqVZNZw/QG9RC/AyUDNhhXjQDKwOnvQ2gfpyHVvXa
/AjsuEyXzTj1D2aKu1HB8j1+4rSUb+M/KpbwX+7t8TG8rXQwkdZlwHkqhlq8sP9qdaZ2k/hDxgD/X5dsl5Xm
Yrh+EV9XvHrTf5bBH9QncGefjVq6lB1PWOmebNi+CwuCIZLgHRfUhFZq4N5/Agi6C6WExK8SvPzm7BM8hV1Y
aANwQx5JhaaZ7htYmEwqR1/W+9tkCOyUxwJZe2AWpkBoAFxVHn7XWhHx68X67gpC6a2uPq7l9EGc6mDvdZ4Br
Z5kwe6mhGMbrX3G+d5iT1tthQuIbA4/S52s28M3d4U9y7uqWb/m+8ySRFAO1BDN0smlV/JBqJl6vuvmyBfQv
p9mbeAPzCeWkGIgu5vpZAU4UtASajfpdeHi+ER7omPzh6BgLRu1+3XeYk4lQ7Z9mr5+CC6jPjl4dB2S2ejM2sxX
NnQ3Q7iXZ7eoz2lfP9kbBPf6Wnh/ZwDPhQ9dzVdQF16sFlHrf66nDz61Wal43X/fSMyzv8ZEaq3ZWqVhAtfrKXL
T2dD6jn/Qdvjx3xXrK+rQq06e9/nXWVOB8bwg/ZWF6/cfqCrB/xFf4Q7thk5WbtNJ1FLQyxs9hiTWj/F8tHr7I
+cG1DX0//kc+zPgMlG+OtZwNV6ivPsIYg1pQqNP86AOlExmjDQCIvVY7xc1TX/7HKfJzjZ6+b1k7Hs94sw
Fm1DYvR1h4KQD5o6T1GyPiyzAPfnQZ1ZeVm7G2Y+tO3PUoN/l1h2zjiBEDn1udt7LuoR7k+XhKY+HqOOthI
SXxHNbxS5oFd7n/68e6GVtAWl1L/UXf1bc+Kxgo2u7g4x6kRNK6DqTVUZyDRHaSIDAVVLPMQw9cS8bNUlVSv/
rdA2kpQWoWXB+a88Na5awHOLo5NP/cz6H5Wb6YJpXmP+0aa/4DRv1RTRYaA1GnaO5Un0ddYAP+4tPyMFFyFSU
dz8R2Uxv7ESkG/cppV3Z/fbBvNcGpQN+RSXeLpEtuHYtRtfWJ3/cw6FRTV6OEHQTM2NKu9wt4ABDmTQu47oup
RAGfl9JUz8JIxmuPjGvLEp9Zr3R/6HSCbNJ8b8zC/Z1Zp5UONR/MAhFupzG7xWsZ7P63MGc6tFQA8hVSf9c4
wQPbo6vJxL58ElsWJMaVKr3Rao9UC3sZ6B1a5kKYZ8OB1B8HEC0e5HPeZ+BL9z7VhtGp3JzEk3qIEgDo0dnE
Kx/yNmGTD6wjcfS0OtVVm90hfY8W59tL4sHtrvJNmFTYbwBb/p7pQXeITP+4/MHx4m5aV3JN0YkeZC1gvWZK
dLbPrKq6rRqVp9KYyt/5hONyM4jL9MAggzVR+2g6ecEhxU5ES7U2vQ1prIRKozjk/nyxmbJLKQ7+ofzzbsQN
pCX/acXdvu9UnrIrnxSqobuwFcmeZrOrf/b5Kfp/Wq9Yi5hf9mkT7pbrMVxHGkyqH+eA9wtPVIzY6PqH2CvU
VJoNMhVpVQV0FZf7Kp+CiRqZEbQ4XnAWDr6h4GaSy5S6MK7exWspHwkx0Rt/kvzA+J6EW6zKTUEG1ghHJV5mAH
sOmu0Y1kpuDAVvRU62ZyVhn+Tz4MiKehzM3Md0bypExiNd0r1ZjMeknmLXO40Q9qvOMx5D5ODgzeOzT4xSU
lwFNi+i0TJYMRJ8Dxjjgd2Ry8l8KTL/L+MNUhTZ1TTzWcTv7rs3ObDcULTDsnNCAWQlt1WqXG3TRFSsxPyk/J
JL1h20yNO4bR79hFa6Ri7BdS0Jzyx9H0dOldEJ/UHf5zNmnHr5uTjtm2cNsa5gNtgS+pT2R/uNxVX+QGCtbkK
Kr7eUsIfCjoQTn5ADf6wbv1kzDoFbCJoVLGTDvHVlwcCJWeEC2wP3iNb2XC7ymcBdk8gTf/dn+vcQ3CgDRhtV9
akGrTJpZCmHjPVBsUHPDubdzOixhmgfpKdTGdffNpVV/w5cGf4s0R+R9ZpGdKsOzzk5BBHeQ+BpWq8+HhBs
k7bmSP9e/mCj/EXwtJ4ngoIaeWB1YqGtbWL1dNCtGfiJPiIGtHh+pTcU9YT5bo/KvGtHDRBeg/oGYzu5wk/
3Bet17ds1XMxvZzwT9Rg9t4Cwov5BfCbOgLYRy4AB1VRDWy+k1z8C3Mjo/oH4PrE7XpGjXhg6N1UBR1GtI4
IWfJYbUvsKPI9kK7xf/r+Bcxzt8bSQn1aDxHv49CwP/TgmdAW+BQ7vGSJ0R9mEGwLOaoYa6M6EvrB4FUjQ
rQWJc7w0LovUuHt8ttZupGbX2ydNrSlAdhMMLGwBeS3cUSyxFOR1+Oo2hDgeXHpC4v+KetV8jFKRtDCSYI8
f1DXL3KGzPPMjIvtwb2PBfgtx6DlNN2v1+2WUjwPvFr8NOWkOSU0nbC1Mke3n9fmJRhMJD/4+cJU0oQdxE0t
6bBeKp7wgd5MTNg9M2HM/ec9UZR+pJrXp8jXqhN1r9yNUtIxlNpwpM+fDU3wLvWdUVnpnNzcvbfpzNTr+Hr6
kVjCOJ+WIm+KAJ/EnjWxDLm+fVE3ZFqzfLTYYQeaX4dLQpUn2xIaBnAlm+Nr3lYB5MqkZY8ZgAhFXXmDAQL2t
wJ1ObeDxJZ+mWHbDrH+AtF/75yR9eD0A9uq0qEBWTNOyS57MhDoxQV0EPdwPfK3s9d9T2Ne+9Kt0tGw8Pq31Z4
mpBxiNB1+OsNjScA9Nl4z+9Px5R3G0sZlP4fpt2Mtn3D8MsCzGg9JhYYdCrhjwZD+Q8CY1sHrn1N5pQN0l+KpJ
x9pePWM+L5NXMx4dd3WkUfM2V5J2vPSiH/HvVtBvkMjzT8sZw4vnRprXkYUK0I9vS3rJ7Uzzn37cgGe2bBdRr
O5b5SlC/nYJu9PxuZh99nP9hd6FU8fuHRFZL18ZuIDEhjSYZJwxv9YDzpScZE+Ys/QHKVI03KBwUzvVzQcmXN
kbpySmzXOKZbjH3a/R+Z3hMt0gmVXJN5sFdZTwT6y+4FQoUt9j+vUN0bY8nzLbS0ZVpuE4lcR8Yd8zRW+Ke+Y3
qaXPlwn4E9bnN6l4uXVNcxOGHsEN8IXsYdQrTnNQkx0LVNeDzvcu61xrB1z+gdw7yT4c6uR1YbDLV1UOtqjs
8uNX7sn5+pqFjD8+5bP5lgKOG1o96rNJyE1jgaO4JdXtTXS0X/9D4MOgrgXEQDg1zrZhzZd4wPYydr1bWD1zPIn
4TWI3/5ZhUvYAeh5NFLUpwtvnFqdlGkXsVKrDSGhDcfPXcVmJGv8TT6ebZvFjgHFsqlZn9u4xczeO2XtlZ
Tp1ubkHRFhKmvd4pOY6zhdt/ozA0K6M1+fn2GWDB6CqK5enW/5xlOzZV+4f7ywOlJ9NznaNM9E1m2kETiu7e
wkEEOlsaj6LBWg4PVG5dYZeYuZoTBEq6N92jJmYF9wQRU5ocwEOB7cfpP3f6jgZY3PishinEj+xgHHtst+0

Implement the *transform* function:

1. Receive an image object and a backward map (a dictionary that maps pairs (y, x) to partial pixels)
2. Create a new empty numpy tensor in a size according to the keys of the backward map
3. Decide the color for every coordinate, by calling *get_average_color* with the partial pixels and the source image
4. Return a new image object

In [8]:
```
def transform(source_image: MyImage, backward_map: dict) -> MyImage:
    # TODO
    return target_image
```

Implement a simple forward scale mapping. Scale **x** by **sx** and scale **y** by **sy**.

In [9]:
```
def forward_scale_mapping(old_y: float, old_x: float, sy: float, sx: float) -> (flo
at, float):
    # TODO
    return new_y, new_x
```

Now implement a backward mapping operator. Given a target image and known scales, where did each pixel translated from? The result may be a "fractional" pixel.

You can implement this function by a single call to *forward_scale_mapping*, think how. Partial points will be given to students who do not call *forward_scale_mapping*.

In [10]:
```
def backward_scale_mapping(old_y: float, old_x: float, sy: float, sx: float) -> (fl
oat, float):
    pass  # TODO
```

We will now test your code for sanity.

In [11]:
```
class ScaleTest(TestCase):
    def test(self):
        y, x = randint(0, 768 - 1), randint(0, 1024 - 1)
        sy, sx = uniform(0.1, 10), uniform(0.1, 10)

        new_y, new_x = forward_scale_mapping(y, x, sy, sx)
        old_new_y, old_new_x = backward_scale_mapping(new_y, new_x, sy, sx)
        self.assertAlmostEqual(y, old_new_y, msg="Diff in y")
        self.assertAlmostEqual(x, old_new_x, msg="Diff in x")

ScaleTest().test()
```

Now let's create a full backward mapping dictionary. Given a source image's dimensions, you need to calculate the target image's dimensions, and then for every target pixel **(new_x, new_y)**, store the source pixel **(old_x, old_y)** in the dictionary.

In [12]:
```
def calculate_all_backward_scale_mapping(source_height: int, source_width: int, sy:
float, sx: float) -> dict:
    """

    :rtype: (dict, int, int)
    :param source_height: Original height of the image in pixels
    :param source_width: Original width of the image in pixels
    :param sy: Forward height resize scale
    :param sx: Forward width resize scale
    :return: A mapping from forward integer pixels (y, x) to the backward fractiona
l pixels (old_y, old_x)
    """

    # TODO

    return backward_map
```

Let's scale the mandrill using 3 sampling methods.

In [13]:
```
sy, sx = 0.9, 2.1

run(mandrill)

old_height, old_width = mandrill.height, mandrill.width
backward_map = calculate_all_backward_scale_mapping(old_height, old_width, sy, sx)

run(mandrill, nearest_neighbour, backward_map)
run(mandrill, bilinear, backward_map)
run(mandrill, gaussian03, backward_map)
```

```
****************
```

```
PSNR = inf dB
MSE = 0.00
Elapsed: 0.11 seconds
****************
****************
nearest_neighbour
```

```
Elapsed: 5.19 seconds
****************
****************
bilinear
```

```
Elapsed: 5.53 seconds
****************
****************
gaussian 0.3
```

```
Elapsed: 196.56 seconds
****************
```

Part 3: Rotation Transformation

Implement a simple forward scale mapping. Given **x** and **y**, translate them (use the **height center** and **width center** as the origin) and rotate them using the **cosine of t** and the **sine of t**.

In [14]:
```
def forward_rotation_mapping(old_y: float, old_x: float, height_center: float, width_center: float, cost: float, sint: float) -> (float, float):
    # TODO
    return new_y, new_x
```

Now implement a backward mapping operator. Given everything, where did each pixel translated from? The result may be a "fractional" pixel.

You can implement this function by a single call to *forward_rotation_mapping*, think how. Partial points will be given to students who do not call *forward_scale_mapping*.

In [15]:
```
def backward_rotation_mapping(new_y: float, new_x: float, height_center: float, width_center: float, cost: float, sint: float) -> (float, float):
    pass  # TODO
```

We will now test your code for sanity.

In [16]:
```
class RotationTest(TestCase):
    def test(self):
        height, width = randint(1, 768), randint(1, 1024)
        y, x = randint(0, height - 1), randint(0, width - 1)
        t = uniform(0, 360)

        cost, sint = cos(t), sin(t)

        height_center = (height - 1) / 2
        width_center = (width - 1) / 2

        new_y, new_x = forward_rotation_mapping(y, x, height_center, width_center,
cost, sint)
        old_new_y, old_new_x = backward_rotation_mapping(new_y, new_x, height_cente
r, width_center, cost, sint)
        self.assertAlmostEqual(y, old_new_y, msg="Diff in y")
        self.assertAlmostEqual(x, old_new_x, msg="Diff in x")

RotationTest().test()
```

Now let's create a full backward mapping dictionary. Given a source image's dimensions, you need to calculate the target image's dimensions, and then for every target pixel **(new_x, new_y)**, store the source pixel **(old_x, old_y)** in the dictionary.

In [17]:
```
def calculate_all_backward_rotation_mapping(height: int, width: int, t: float) -> d
ict:
    """

    :rtype: dict
    :param height: Height of the image in pixels
    :param width: Width of the image in pixels
    :param t: Forward rotation angle in degrees (in relation to the center of the i
mage)
    :return: A mapping from forward integer pixels (y, x) to the backward fractiona
l pixels (old_y, old_x)
    """

    # TODO

    return result
```

Let's begin rotating the mandrill 10 times, 36° each, with our sampling methods. We expect to see unwanted artifacts near the edges, because the image is not padded.

In [18]:
```
t = 36
assert 360 % t == 0
iters = 360 // t

run(mandrill)

backward_map = calculate_all_backward_rotation_mapping(mandrill.height, mandrill.wi
dth, t)

run(mandrill, nearest_neighbour, backward_map, num_iters=iters, gif=True)
run(mandrill, bilinear, backward_map, num_iters=iters, gif=True)
run(mandrill, gaussian03, backward_map, num_iters=iters, gif=True)
```

```
****************
```

```
PSNR = inf dB
MSE = 0.00
Elapsed: 0.08 seconds
****************
****************
nearest_neighbour

<IPython.core.display.Image object>

PSNR = 16.21 dB
MSE = 1557.86
Elapsed: 16.86 seconds
****************
****************
bilinear

<IPython.core.display.Image object>

PSNR = 16.76 dB
MSE = 1371.52
Elapsed: 18.77 seconds
****************
****************
gaussian 0.3

<IPython.core.display.Image object>

PSNR = 16.76 dB
MSE = 1371.72
Elapsed: 130.53 seconds
****************
```

Implement the ***pad*** function. It should take an image and return a larger image, having the original image in the center and white pixels around it, such that no pixels will be lost during a rotation operation.

1. Hint: Where would a corner pixel translate under a 45 degrees rotation? Think of a square source image for example.
2. Hint: Should the target image be square? Is there symmetry in the probem?

Partial points will be given for a larger padding than needed.

In [19]:
```
def pad(image: MyImage) -> MyImage:
    # TODO
    return MyImage(new_tensor)
```

In [20]:
```
padded_mandrill = pad(mandrill)

run(padded_mandrill)

backward_map = calculate_all_backward_rotation_mapping(padded_mandrill.height, padd
ed_mandrill.width, t)

run(padded_mandrill, nearest_neighbour, backward_map, num_iters=iters, gif=True)
run(padded_mandrill, bilinear, backward_map, num_iters=iters, gif=True)
run(padded_mandrill, gaussian03, backward_map, num_iters=iters, gif=True)
```

```
****************
```

```
PSNR = inf dB
MSE = 0.00
Elapsed: 0.08 seconds
****************
****************
nearest_neighbour

<IPython.core.display.Image object>

PSNR = 164.01 dB
MSE = 183.12
Elapsed: 43.55 seconds
****************
****************
bilinear

<IPython.core.display.Image object>

PSNR = 165.10 dB
MSE = 142.63
Elapsed: 50.56 seconds
****************
****************
gaussian 0.3

<IPython.core.display.Image object>

PSNR = 164.89 dB
MSE = 149.70
Elapsed: 354.92 seconds
****************
```

Bonus: load an image of your choice and define your own Gaussian, such that for 10 rotations, the PSNR is higher and the MSE is lower in comparison to bilinear. Attach the image to your submission.

Partial points will be given for unsuccessful attempts.

In [21]:

```
# The Gaussian's standard deviation
my_std = 1.0
# The distance of the regular grid's sampling
my_d = 1.0
# The epsilon to add to the sampling result
my_epsilon = 1.0

my_gaussian = partial(parametric_gaussian, std=my_std, d=my_d, epsilon=my_epsilon)
my_gaussian.__name__ = 'my gaussian'

my_image = PILImage.open("path/to/your/image.jpg").convert("RGB")
my_image = MyImage(array(my_image))
my_image = pad(my_image)

run(my_image)

backward_map = calculate_all_backward_rotation_mapping(my_image.height, my_image.wi
dth, t)

run(my_image, bilinear, backward_map, num_iters=iters, gif=True)
run(my_image, my_gaussian, backward_map, num_iters=iters, gif=True)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-21-83f8fd531c8c> in <module>
      9 my_gaussian.__name__ = 'my gaussian'
     10
---> 11 my_image = PILImage.open("path/to/your/image.jpg").convert("RGB")
     12 my_image = MyImage(array(my_image))
     13 my_image = pad(my_image)

~\Anaconda3\envs\python37new\lib\site-packages\PIL\Image.py in open(fp, mode)
   2764
   2765     if filename:
-> 2766         fp = builtins.open(filename, "rb")
   2767         exclusive_fp = True
   2768

FileNotFoundError: [Errno 2] No such file or directory: 'path/to/your/image.jpg'
```