1 Introduction

Hello curious person! In this document I will try to informally explain some of the concepts and the coding seen on my demos page. This is done in my free time (of which I have very little) so it may or may not contain everything posted. Moreover, there's bound to be some small errors here and there so I wouldn't exactly use this as reference material.

2 Monte Carlo Integration

2.1 The Mathematics of the Monte Carlo Method.

Anyone whose taken Calculus knows that integrating is a non-trivial computational task. Monte Carlo integration is an approach to numerical integration done by collecting random samples of the function. So how does this approach work? At the tail end of your calculus course you may have learned the following formula for the average value of a function over an interval $[a, b]$:

$$f_{\text{avg}} = \frac{1}{b-a}\int_a^b f(x)\,dx$$

This formula certainly looks reasonable if you're familiar with the definition of the Riemann integral.

$$\int_a^b f(x)\,dx := \lim_{n\to\infty}\sum_{j=0}^{n} f(x_i^*)\,\Delta_i.$$

Here, Δ_i is the length of the sub-interval from where we sampled the function. If we assume that all the interval lengths are equal, then $\Delta_i = (b-a)/n$ where $(b-a)$ is the length of the interval we are integrating over. The above formula for the average looks reasonable because we essentially expect it to be a limit of the average value of samples as the number of samples increases.

$$\begin{aligned}
f_{\text{avg}} &:= \lim_{n\to\infty}\frac{1}{n}\sum_{j=0}^{n} f(x_i^*) \\
&= \lim_{n\to\infty}\sum_{j=0}^{n} f(x_i^*)\frac{1}{n} \\
&= \lim_{n\to\infty}\frac{1}{b-a}\sum_{j=0}^{n} f(x_i^*)\frac{(b-a)}{n} \\
&= \frac{1}{b-a}\lim_{n\to\infty}\sum_{j=0}^{n} f(x_i^*)\Delta_i \\
&= \frac{1}{b-a}\int_a^b f(x)\,dx
\end{aligned}$$

If you look closely at the derivation above, this gives insight on how the Monte Carlo method works. Specifically,

$$\int_a^b f(x)\,dx = (b-a)\cdot f_{\text{avg}} = (b-a)\cdot\lim_{n\to\infty}\frac{1}{n}\sum_{j=0}^{n} f(x_i^*).$$

If we take random samples, the law of large numbers says that the limit will approach the correct value. In practice, after taking a large n, you'll get an estimate for the value of the integral with error bars. The size of the error bars is related to the sample variance, but its best to ask your local statistician/probablist for a more concrete description.

2.2 The Code for the Monte Carlo Method

Lets take a look at the code for `montecarlo.py`.

```
import random
import math
import numpy as np
import matplotlib.pyplot as plt

def f(x):
        return math.exp(x)

n = 100 #number of samples

x = np.linspace(0,1,50)

Domain = [i for i in x]
Range = [f(i) for i in x]
Sample = [random.random() for i in range(n)]
FSample = [f(i) for i in Sample]

Average = sum(FSample) / n
print(Average)

scatter = plt.scatter(Sample, FSample, color='red')

plt.plot(Domain, Range)
plt.show()
```

The output of this script will be a plot and the value of `Average=sum(FSample)/n`.

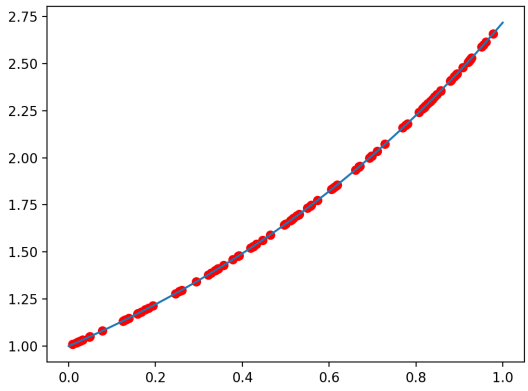


Figure 1: Output of the script `montecarlo.py`

The plot and the value will be different each time the user runs the script since they samples are done (pseudo)randomly. The user can specify the function to be integrated by changing the return value of the function `f(x)` and the value of `n` determines the number of samples being taken. Part of the script is dedicated to simply plotting the function and the other part is for computing the value of the integral via the Monte Carlo method. To plot the function, we specify a line space `x = np.linespace(0,1,50)`. Here the line space is from 0 to 1 and 50 steps are going to be taken in the interval of $(0, 1)$. The list `Domain` is simply recording the steps and `Range` is a list of the function `f(x)` on the steps.

Example 1. Let's change this slightly to see an example.

Input:

```
x = np.linspace(0,1,5)
Domain = [i for i in x]
```

Output:

```
[0.0, 0.25, 0.5, 0.75, 1.0]
```

The part of the code used to calculate the average of the random sample is given by:

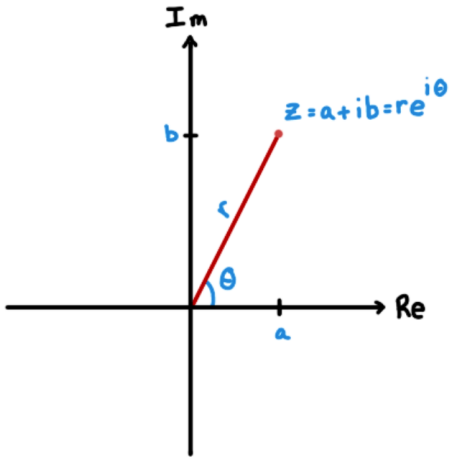
```
Sample = [random.random() for i in range(n)]
FSample = [f(i) for i in Sample]

Average = sum(FSample) / n
```

We are taking a slight advantage of the fact that the interval we are interested in is $(0, 1)$. Specifically, `random.random()` will give pseudo-random values in the interval $[0, 1)$. If we wanted to change the interval we draw random numbers from we'd have to use something like `random.uniform(a,b)` which will pick out uniformly distributed samples over the interval (a, b). It should also be pointed out that the lists were written using the *list comprehension* feature of python. Other than the small details above, the average is printed out and the random samples are plotted using the `plt.scatter`.

3 Plotting Complex Functions

3.1 The Mathematics of Complex Functions

Obviously, we cannot review all of complex analysis here, but what we can do is go over some of the basics needed to understand the output of the code. A complex number z is a number of the form $z = a + ib$ where a and b are real numbers and $i = \sqrt{-1}$. We visualize complex numbers by picturing them on a plane, called the "complex plane", "Argand plane", or "Gauss plane". For this we identify $z = a + ib$ as the Cartesian coordinate (a, b) and we call a the real part of z and b the imaginary part of z. We can also think of a complex number in polar coordinates (r, θ) where r is the distance to the origin and θ is the angle made with the real axis. Likewise every complex number has a polar for $z = re^{i\theta}$ where θ is a real value angle. Before we get into any formulas lets look at a diagram.



This makes sense for the time being but further thought shows there's some difficulty ahead. For instance, how do we define $e^{i\theta}$? This is an easy question to answer if we use the famous Euler Formula (about as famous as any of his other formulas/theorems, Euler has a lot of concepts named after him `https://en.wikipedia.org/wiki/List_of_things_named_after_Leonhard_Euler`)

$$e^{i\theta} = \cos(\theta) + i\sin(\theta).$$

Let's notice a little bit of ambiguity,

$$e^{i\theta} = \cos(\theta) + i\sin(\theta) = \cos(\theta + n2\pi) + i\sin(\theta + n2\pi) = e^{i\theta + n2\pi}$$

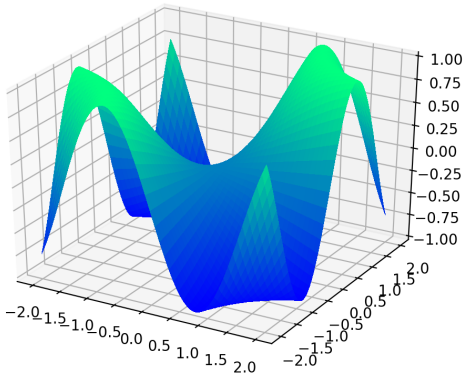
for all $n \in \mathbb{Z}$ since both $\cos(\theta)$ and $\sin(\theta)$ are 2π periodic. Hence, any complex number z has an infinite number of polar forms

$$z = re^{i\theta} = re^{i\theta + n2\pi}$$

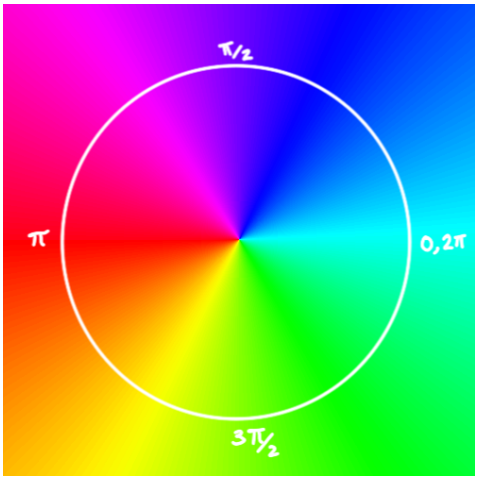
for all $n \in \mathbb{Z}$. When $\theta \in (-\pi, \pi]$ we call that the *principal argument* for z. So given a complex number $z = a + ib$, then for its polar form we have that $r = \sqrt{a^2 + b^2}$ and $\theta = \arctan 2(\frac{b}{a})$. Here, $\arctan 2$ is a specially defined version of the standard arctan function but returns the principal argument. In other words, $arctan2$ returns values in $(-\pi, \pi]$ while $arctan$ returns values in $(-\pi/2, \pi/2)$. A function $f : \mathbb{C} \supseteq A \to B \subseteq \mathbb{C}$ takes as an input a complex number z and outputs a complex number $f(z)$. However, if we're going to visualize this function we have a small problem. We effectively have a two dimensional input $z = a + ib \mapsto (a, b)$ and a two dimensional output $f(z) = c + id \mapsto (c, d)$. Hence, we need four dimensions to plot $f(z)$! For example,

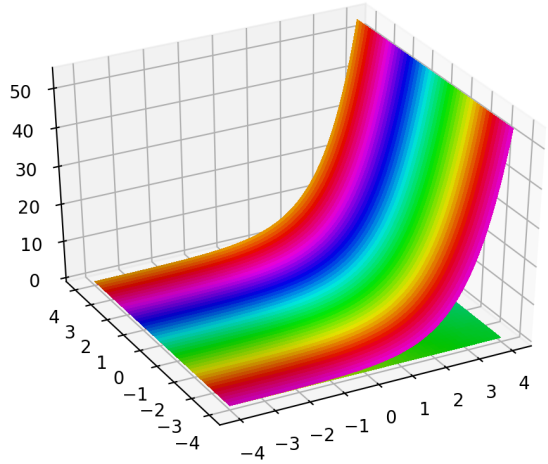
$$f(x, y) = \sin(xy)$$

takes a two dimensional input (x, y) and has a single number as an output, namely $\sin(xy)$. We can plot this as a surface in three dimensions.

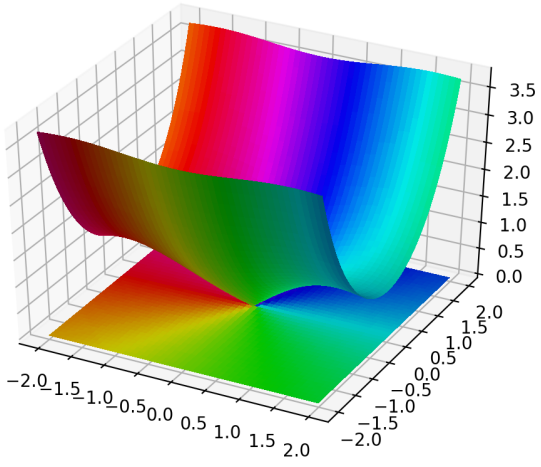


To plot a complex function, we are going to the Cartesian representation for the input and the polar representation for the output. For an input $z = a + ib$ we will represent the output $f(z) = re^{i\theta}$ as a point at the height of r above the point (a, b) and colored according to its principal argument θ. Below will be the color scheme used and a few example plots for the functions $e^z, \sin(z)$ and $\sqrt{z}$.

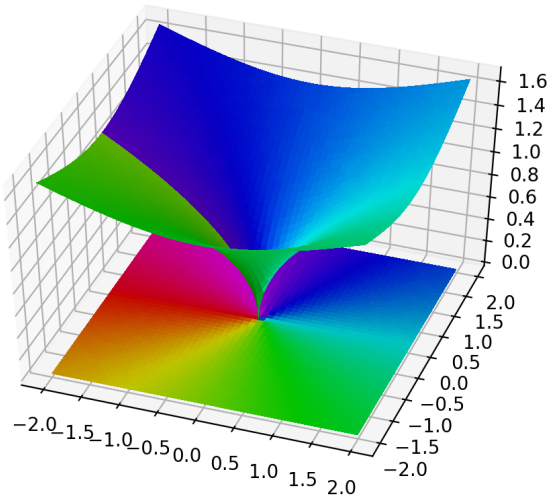




e^z



$\sin(z)$



$\sqrt{z}$

3.2 The Code for the Complex Function Plotter

The interesting part of this code is working with the `meshgrid` function and getting that to work with `cmath`. I'll have the entire code at the end of this section.

Example 2. Lets look at an easy example of what this function does.

Input:

```
r=1
X = np.arange(-r, r, 0.5)
Y = np.arange(-r, r, 0.5)
X, Y = np.meshgrid(X, Y)
```

Output:

```
X = [[-1.  -0.5  0.   0.5]
     [-1.  -0.5  0.   0.5]
     [-1.  -0.5  0.   0.5]
     [-1.  -0.5  0.   0.5]]

Y = [[-1.  -1.  -1.  -1. ]
     [-0.5 -0.5 -0.5 -0.5]
     [ 0.   0.   0.   0. ]
     [ 0.5  0.5  0.5  0.5]]
```

We then get a complex version of the above by using the defined function `z(x,y)`.

```
def z(x,y):
    return x + 1j * y
```

If we print `z(X,Y)` we'll get the following.

```
 Z = [[-1. -1.j  -0.5-1.j   0. -1.j   0.5-1.j ]
      [-1. -0.5j -0.5-0.5j  0. -0.5j  0.5-0.5j]
      [-1. +0.j  -0.5+0.j   0. +0.j   0.5+0.j ]
      [-1. +0.5j -0.5+0.5j  0. +0.5j  0.5+0.5j]]
```

Once we have that, we can then feed this grid of numbers to a complex function using `numpy.vectorize` which deals with the fact that a function in `cmath` does not have arrays as inputs.

```
 f=np.vectorize(cmath.exp)(z)
```

The only other difficult bit is coloring the surface.

```
 #color mapping
norm=mpl.colors.Normalize(-np.pi,np.pi)
my_col = cm.hsv(norm(np.arctan2(f.imag,f.real))) #hsv is a cyclic color map
floor_col = cm.hsv(norm(np.arctan2(Im,Re)))
```

Here, we are using `Normalize` to squish the output of `arctan2` into the range $[0, 1]$. This needs to be done since the colormap `hsv` expects an input within that range.

Here is the entire code for the complex function plotter.

```
import cmath
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')

# Make data.
r=4 #make a 2r x 2r square around 0.
X = np.arange(-r, r, 0.01)
Y = np.arange(-r, r, 0.01)
X, Y = np.meshgrid(X, Y)

def z(x,y):
    return x + 1j * y
z = z(X, Y) #makes an grid of complex numbers

def zbar(x,y):
    return x - 1j * y
zbar = zbar(X, Y)

Im = Y #renaming for clarity
Re = X #ditto

#everything is computed on a grid of numbers.
f=np.vectorize(cmath.exp)(z) #cmath does not get along with arrays, this fixes that.
imf=f.imag
ref=f.real
Mod=np.sqrt(f.imag**2+f.real**2)

#color mapping
norm=mpl.colors.Normalize(-np.pi,np.pi)
my_col = cm.hsv(norm(np.arctan2(f.imag,f.real))) #hsv is a cyclic color map
floor_col = cm.hsv(norm(np.arctan2(Im,Re)))

surf = ax.plot_surface(X, Y, Mod, facecolors = my_col, linewidth=0, antialiased=False)
#surface color
bottom = ax.plot_surface(X, Y, 0*X, facecolors = floor_col, linewidth=0, antialiased=False)
#floor color for ref

# Customize the z axis.
maxvalue=np.amax(Mod)
ax.set_zlim(0, maxvalue)

plt.show()
```

4 Low Pass Filters via the Fourier Transform

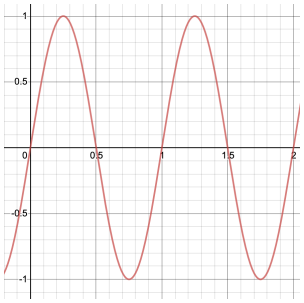
4.1 The Mathematics of the Fourier Transform

The Fourier transform of a function $f : \mathbb{R} \to \mathbb{C}$ is defined via the integral equation

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i t\omega}\, dt.$$

The Fourier transform takes a function over a time domain t to a function over the frequency domain. This process is reversible and to undo the Fourier transform we take the Inverse Fourier transform. In some sense, $\hat{f}(\omega)$ is considered to be a measure of how much the frequency ω is present in the original signal $f(t)$. This is a bit ambiguous, so let's look at a concrete example.

Example 3. The function $f(t) = \sin(2\pi \cdot t)$, depicted below:



This function obviously oscillates at "1 Hz". So this function is oscillatory at only one frequency $\omega = 1$ and we should expect this to be reflected in its Fourier transform. For $f(t) = \sin(2\pi \cdot t)$ its transform is given by

$$\hat{f}(\omega) = \frac{\delta(\omega - 1) + \delta(\omega + 1)}{2i}.$$

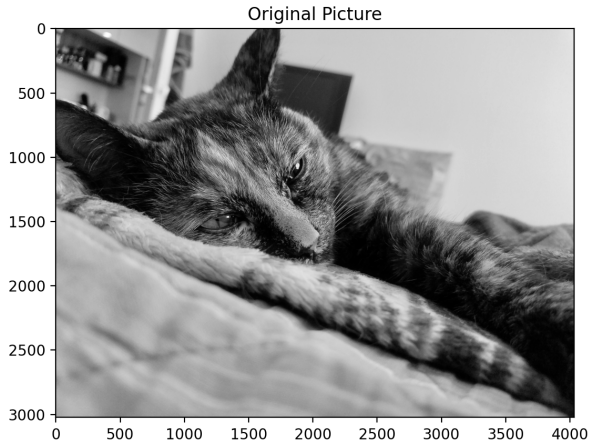
The function $\delta(\omega - a)$ signifies the Dirac delta, which has an "infinite spike" at $\omega = a$ and is zero everywhere else. Ignoring the fact that $\hat{f}(\omega)$ is a complex valued function we see that this function indeed has a spike at $\omega = \pm 1$.

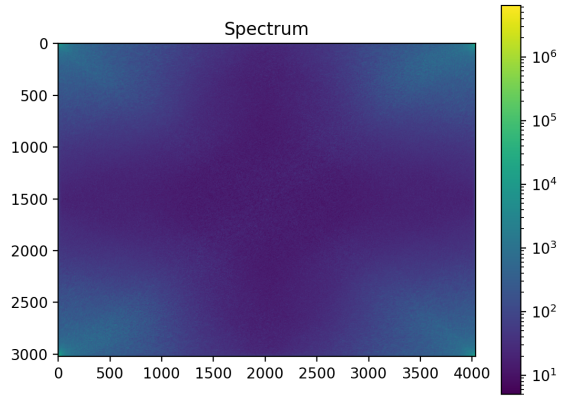
Now, what does this have to do with photos? Plainly, noise in a photo is high frequency. So the general principal is that we take the Fourier transform, prune off the high frequency part, then undo the Fourier transform to recover the original.

Before we discuss how to do this with a digital photo, we have to talk about what kind of data is in a .png file. For our purposes with each pixel in a .png file, there is a vector (array) $p = \begin{bmatrix} r & g & b & a \end{bmatrix}$ corresponding to that pixel's rgba values. The first three letters of rgba have the obvious meanings. The last part, "a" corresponds to the pixel's alpha channel, which is just a fancy way of saying opacity. Since, pixels are discrete objects, we are going to use the discrete Fourier transform (DFT):

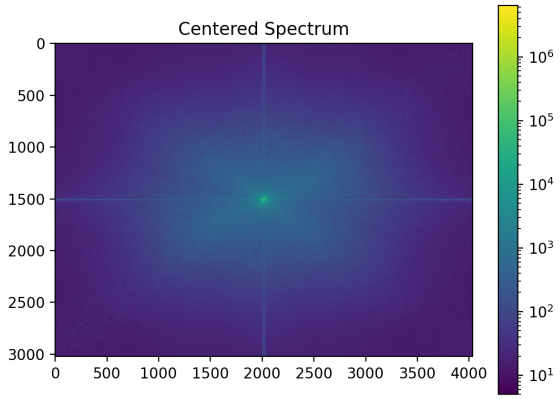
$$\hat{f}(j,k) = \sum_{p=0}^{N-1} \sum_{q=0}^{N-1} f(p,q) e^{-i2\pi(\frac{jp}{N} + \frac{qk}{N})}.$$

Here f : (pixel in spot)(j,k) $\to$ scalar value, where the value is one of the pixels rgba values and the photo is of size $N \times N$. This means we'll have to work with each separate layer (ignoring the opacity layer since that does not need to be filtered). There is also a formula for the inverse discrete Fourier transform, but its similar and we won't actually need it for the code. Instead, we will employ something called the *Fast* Fourier transform, which is an algorithm to calculate the Fourier transform and its inverse ... quickly. For those of you that are curious, it goes from $O(N^2)$ to $O(N \log(N))$. Below is a gray-scale picture and its the result after we take the Fourier transform.

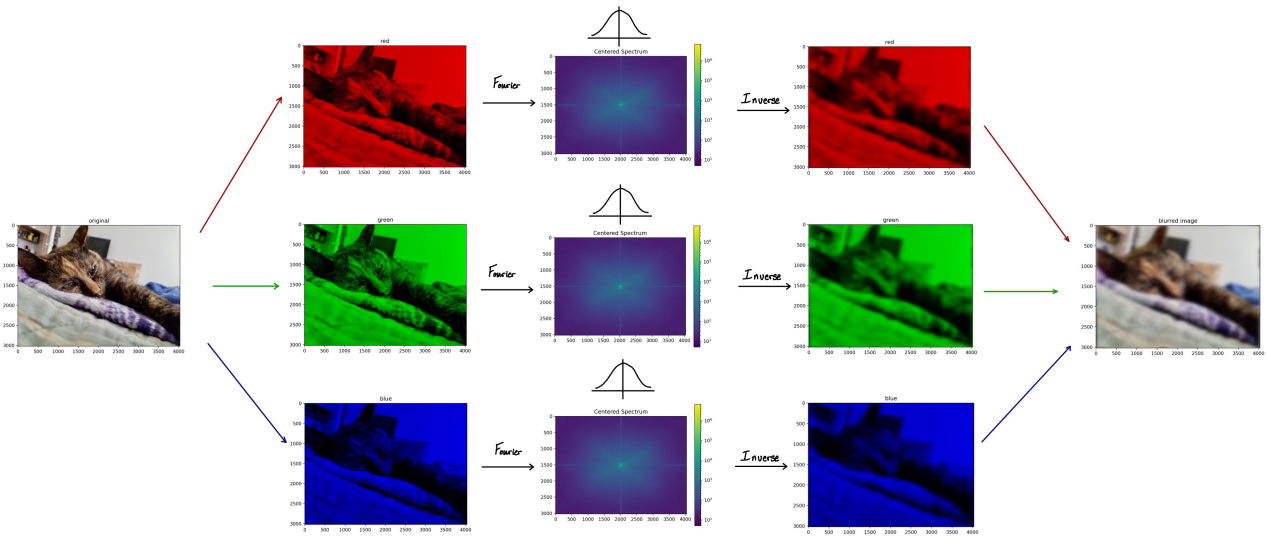




We are not actually ready yet to apply the filter. Before we do that, the Fourier transformed needs to be re-centered since the low frequency data is in the corners. So we re-center the spectrum and place a Gaussian with its peak occurring at the center.



On a slightly less hand-wavy level, what will happen is that each entry of the 2d array that represents the photo will be multiplied by a the value of the Gaussian at that location. The farther away from the center the smaller the value of the Gaussian. Here is a diagram of the entire process.



4.2 The Code for the Low Pass Filter

Time to start talking about the code. The entire code will appear at the end. In the previous section I said we would need to deal with the opacity/alpha channel separately from the red, blue, and green channels. Here are the parts of the code that deal with the opacity channel.

```
def opacity(rgb):
    return rgb[:, :, 3]
.
.
.
opacity_channel = opacity(img)
opacity_part = np.zeros(img.shape)
opacity_part[:,:,3] = opacity_channel
```

We've written a function, `opacity()` which takes out the opacity information from the vector $\begin{bmatrix} r & g & b & a \end{bmatrix}$ for each pixel of the photo. In most cases, its going to be 1 since its opaque. We then make an array of zeros the same size as the the original photo and set the opacity entry to be the data we've taken from the original photo. Again, at the end we should have that each pixel has the vector $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$, assuming the photo has full opacity. For the rest of the channels, we will loop over them with the following code:

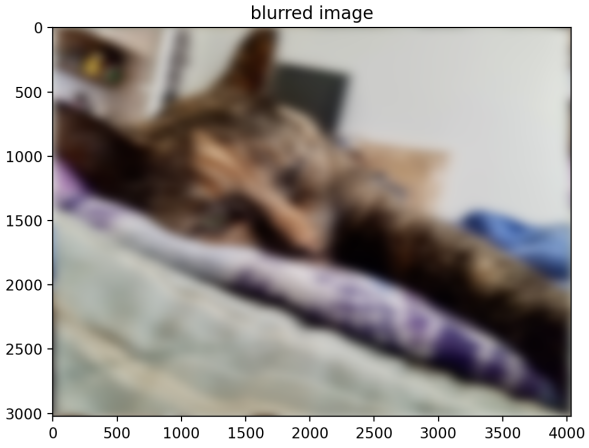
```
for i in range(3):
        channel = img[:, :, i]
        filtered = Filter(channel, 20, 20)
        part = np.zeros(img.shape)
        part[:,:,i] = abs(filtered)
        result = result + part
```

In the loop for each channel, we are applying our `Filter()` function, storing it an array and then adding it to the previous. At the very end, we'll add in the opacity part. The last bit we need to discuss is this `Filter()` function. It's really just the code for the diagram up above.

```
def Filter(channel, s1, s2):
        ft = np.fft.fft2(channel)

        cft = np.fft.fftshift(ft)

        cx, cy = floor(cft.shape[0]/2), floor(cft.shape[1]/2)

        sigx, sigy = s1, s2

        Gauss = np.ones((cft.shape[0], cft.shape[1]))
        for i in range(len(Gauss)):
                for j in range(len(Gauss[i])):
                        Gauss[i,j] = exp(-(((i-cx)/sigx)**2 + ((j-cy)/sigy)**2))

        filtered = np.fft.ifft2(Gauss*cft)

        return filtered
```

The first two lines of this function are dedicated to taking the Fast Fourier transform and then re-centering the data. To do this we employ the functions `fft.fft2()` and `fft.fftshift()`. The values `cx,cy` are defined to be the x and y coordinates for the center pixel of the photo. We are then iteratively building an array called `Gauss` whose entries are given by the formula `exp(-(((i-cx)/sigx)**2 + ((j-cy)/sigy)**2))`. Since, this is of the form $f(x) = e^{-(x-c)^2}$ the further away from c that x the smaller the value of the function. The Gauss array is then multiplied by the array for the centered Fourier transform and then finally the inverse transform is taken by `fft.ifft2()`. The only other bit we need to mention, is that in principle the Fourier transform and its Inverse

returns a *complex* valued function. Hence the presence of `abs()` since we'll get complex data and we want real-valued data. We would even get complex valued data if all we did was take the transform and its inverse due to machine imprecision.



```
import numpy as np
from math import floor
from math import exp
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

#lets us deal with the opacity/alpha channel separately.
def opacity(rgb):
    return rgb[:, :, 3]

#this function will scoop up the process of transforming, centering,
#applying the filter, then untransforming.
def Filter(channel, s1, s2):
        ft = np.fft.fft2(channel)

        cft = np.fft.fftshift(ft)

        cx, cy = floor(cft.shape[0]/2), floor(cft.shape[1]/2)

        sigx, sigy = s1, s2

        Gauss = np.ones((cft.shape[0], cft.shape[1]))
        for i in range(len(Gauss)):
                for j in range(len(Gauss[i])):
                        Gauss[i,j] = exp(-(((i-cx)/sigx)**2 + ((j-cy)/sigy)**2))

        filtered = np.fft.ifft2(Gauss*cft)

        return filtered

#read in image
img = mpimg.imread('yourimage.png')

#initializing final result.
result = np.zeros(img.shape)

#handle the alpha channel separately.
opacity_channel = opacity(img)
opacity_part = np.zeros(img.shape)
opacity_part[:,:,3] = opacity_channel

#We're going to loop over the color channels.
#We will also use the same values for s1 and s2 for all.
for i in range(3):
        channel = img[:, :, i]
        filtered = Filter(channel, 20, 20)
        part = np.zeros(img.shape)
        part[:,:,i] = abs(filtered)
        result = result + part

result = result + opacity_part

#display the results.
plt.figure()
plt.title('blurred image')
plt.imshow(result)
plt.figure()
plt.title('original')
plt.imshow(img)
plt.show()
```