



# Image Deblurring

**Lab Objective:** *Image deblurring and denoising are common operations in image processing, technique widely used in signal processing which shows up in optical, medical, and astronomical applications. It is often a vital step towards detection of patterns such as abnormal tissues or the surface details of distant planets. In this lab, we will delve into associated computational methods of image deblurring. Specifically, this lab covers taking a blurred image restoring it to an unblurred state. We will explore image convolution and deconvolution to gain intuition for image processing before applying more robust variational calculus algorithms .*

## Image Kernels

*Image kernels* are matrices that when convolved with an image perform some useful function. They can detect edges, calculate the gradient, sharpen, denoise, and blur the image. Kernels are flexible and powerful ways to execute functions on an image but they have limitations and can be computationally expensive on larger images. For the purpose of this lab we will focus on a subset of kernels which represent point spread functions (PSF) which can be used to describe both motion and focus blurs.

In image processing, blurs can be represented as what is more commonly known as *convolution kernels* or a filter. Kernels are square matrices of dimensions  $n \times n$ , where

$$n = (r + 1 + r)$$

and  $r$  is generally known as the radius of the kernel. This gives the kernel matrix a central pixel. For example, if a kernel has a radius of  $r = 2$ , each pixel is affected by its neighbors no farther than 2 pixels away and would produce a  $5 \times 5$  matrix.

## Point-Spread Functions (PSF)

*Motion blur* arises from rapid mechanical motions of either target objects or imaging device during image acquisition processes. When the relative motion between an imaging device and its targeted objects is intense enough, the image of an object point could travel many pixels wide on the imaging plane during a single exposure. The object will be spread across several sensors instead of a single sensor resulting in a "fuzzy" image. This spreading and mixture of spatial information are what produces motion blurs. *Focus blur* occurs during optical acquisition where particles in the air disrupt

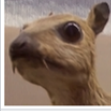
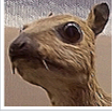

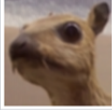
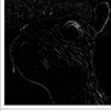
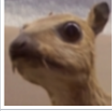

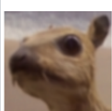
Operation	Kernel	Image result	Operation	Kernel	Image result
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$		Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$		Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$		Gaussian blur 3 × 3 (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$		Gaussian blur 5 × 5 (approximation)	$\frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	

Figure 1.1: Examples of different kernels. This lab will focus on a modified Gaussian blur.

the light ray patterns causing optical turbulence or from an object being outside the depth of field of the camera lens.

The *point-spread function* (PSF) is a more specific kernel. The  $PSF(x, y)$  is a function  $f : \mathbb{R}^2 \rightarrow \mathbb{R}$  or in the case of images, a finite/discrete real-valued two dimensional matrix. For example,  $PSF(x, y) = 0.2$  means that the point  $P(a, b)$  sends 20% of its information to point  $Q(a + x, b + y)$ .

PSFs can help synthetically model motion and focus blur. In Figure 1.1, there is an example of a box blur. Here a pixel is affected by all of its surrounding neighbors. To recreate a synthesized motion blur, we need to model similar information that is spread across many pixels. Instead of a pixel being affected by its surrounding neighbors, we need the pixels to be affected by one or a string of neighbors next to it (depending on the radius).

We can also choose which direction the motion blur occurs, by applying a rotation matrix to our kernel. This function is available in the OpenCV library. OpenCV (Open Source Computer Vision Library) contains a set of optimized machine learning and computer vision algorithms for image processing. To use OpenCV in this lab, it needs to be installed using the following command in the terminal.

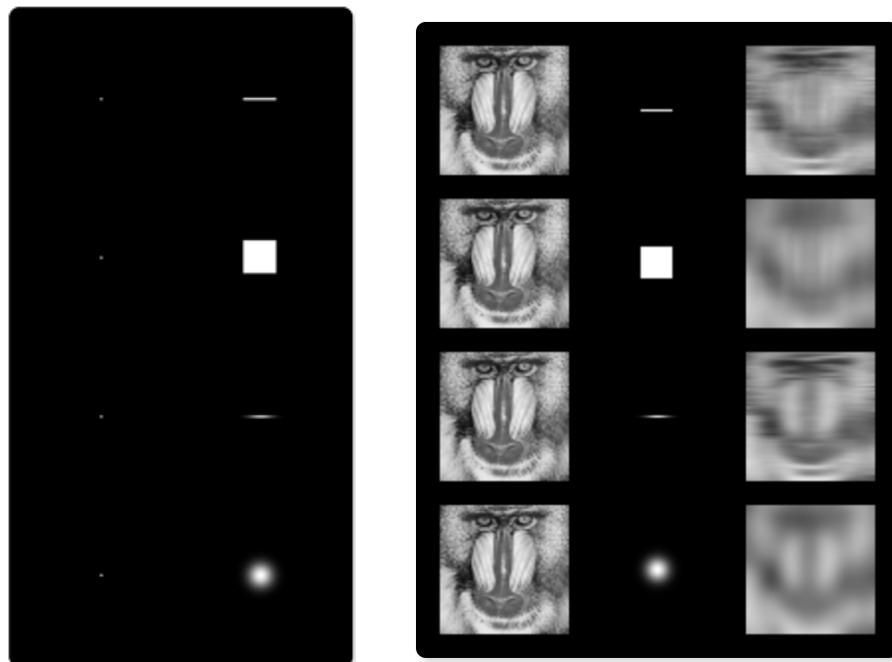
```
pip install opencv-python
```

Use the code below to rotate the motion blur.

```
import cv2
# Theta gives the angle to rotate our motion blur.
rotation = cv2.getRotationMatrix2D(center = (r,r), angle = theta, scale = 1.0)
# The code below applies the rotation to the kernel.
kernel = cv2.warpAffine(kernel, rotation, dsize = (n,n))
```

To synthetically model a focus blur, we need something that forms a circular contour around a single point. In this case we will use the Gaussian kernel, which is defined as

$$G(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2}\right)$$



PSFs increased to show detail

PSFs applied to an image

Figure 1.2: PSFs: 1D box, 2D box, 1D Gaussian, 2D Gaussian

Here  $\mathbf{x}$  is fixed and we can move  $\mathbf{x}'$  in any direction, because the kernel is circular. The  $\sigma$  is the kernel width parameter, a large  $\sigma$  would imply a smoother fit and vice versa.

**Problem 1.** Write two PSF generators, namely the Gaussian blur kernel and the motion blur kernel.

```
def motion_psf(radius, theta):
    """
    Sets up a convolution/deconvolution kernel based on kernel size and ↔
    rotation angle
    Parameters:
    -----
    radius: int,
        radius of the kernel

    theta: float,
        angle of the rotation in degrees in degrees
        positive values are counter-clockwise rotation
        negative values are clockwise

    Returns:
    -----
```

```

kernel: ndarray of shape (n, n),
        numpy array of the motion blur kernel
        n must be odd
"""
pass

```

```

def focus_psf(radius=2, sigma=1.):
    """
    Creates a Gaussian kernel based on kernel size and choice of sigma
    Parameters:
    -----
    radius: int,
            denotes the radius of the kernel
    sigma: float,
            kernel width parameter
    Returns:
    -----
    kernel: ndarray of shape (n, n),
            numpy array of the Gaussian blur kernel
    """
    pass

```

## Convolution

*Convolution* is a mathematical operation on two function that produces a modified version of one of the original function. The equation is defined as the integral of the product of two functions where one is reversed and shifted. The continuous case is found below in the following the form

$$g(t) = \int_{-\infty}^{\infty} f(\tau)k(t - \tau)d\tau$$

The discrete case of the convolution is given by

$$g(t) = \sum_{-\infty}^{\infty} f(\tau)k(t - \tau)d\tau$$

Similar to the above one-dimensional case, the following represents an n-dimensional bounded convolution in the discrete case

$$g(t_1, t_2, \dots, t_n) = \sum_{\tau_1=-M_1}^{M_1} \sum_{\tau_2=-M_2}^{M_2} \cdots \sum_{\tau_n=-M_n}^{M_n} f(\tau_1, \tau_2, \dots, \tau_n)k(t_1 - \tau_1, t_2 - \tau_2, \dots, t_n - \tau_n)$$

Image processing uses the 2-dimensional bounded discrete equation, where  $g$  is some filtered image and,  $f$  is some image that we wish to recover, and  $k$  is the kernel. The operation we wish to use to convolve the image and the kernel is a naive algorithm that applies the same PSF to all of the pixels of the image. More specifically, the convolution take two discrete real-valued matrices

(a luminance image and a convolution kernel) and makes the center of the kernel slide along each pixel in the image. At each pixel, the kernel is multiplied pointwise with all the pixels it covers. The next step is to take the sum of these products and scale the result so that it falls between the 0-255 greyscale range. The rescaled values is then used to replace the original pixel value.

We also need to consider how to approach computing the edge pixels in the convolution function. Some methods include duplicating edge pixels or ignoring the edge pixels and only accounting for pixels that are completely surrounded by neighbors. Another method involves padding the image with zeros. Use the code below for padding, the lab will later cover how this satisfies the boundary conditions for the total variation problem.

```
# Initialize the Source image (src) and the number of padded pixels.
padded_img = cv2.copyMakeBorder(src, top, bottom, left, right)
```

**Problem 2.** Create a `convolve` function that takes in an image and a kernel. Pad a copy of your original image and rescale the convolved values, so that they fall between the 0-255 greyscale range.

```
def convolve(image, kernel):
    """
    Modifies an image using the kernel
    Parameters:
    -----
    image: ndarray of shape (n, m),
           a greyscale image
    kernel: ndarray of shape (d, d),
            numpy array of a blurred kernel
    Returns:
    -----
    convolved_image: ndarray of shape (n+p, m+p),
                     padded image array that has been convolved with the kernel
                     p indicates the number of pixels that are padded onto the image
    """
    pass
```

**Problem 3.** Implement the `convolve` function that was made in Problem 2 using a kernel from your `motion_psf` and `balloons_resized_bw.jpg`.

## Blurring as Convolution

Now that we have some intuition about convolution and a naive algorithm, we want to introduce the *Fast Fourier Transform* (FFT). The FFT gets its name because the speed it takes to convolve two functions is faster than a standard convolution. Usually if the size of the filter kernel is large, then using the FFT would give greater stability and speed.

To implement this method of convolution, we need to apply the Fourier Transform which simplifies the convolution from a double sum to elementwise multiplication. The convolution in the Fourier Transform becomes the operation below

$$\mathbf{F} \cdot \mathbf{K} = \mathbf{G}$$

Here, F, K, and G are the Fourier Transforms of f, k, and g respectively. Finally, the inverse Fourier Transform of G would be taken to receive the convolve image.

```
def convolve_fft(image, kernel):
    # Apply fft on image and kernel
    image_fft = np.fft.fftn(image)
    kernel_fft = np.fft.fftn(kernel)

    # Convolve
    matrix_fft = np.fft.ifftn(image_fft*kernel_fft)

    # Rescale convolved image
    convolve_img = matrix_fft/np.amax(matrix_fft)
    return convolve_img.real
```

## Deconvolution

Unwanted convolution is an inherent problem in transferring information. Deconvolution is the process of filtering an image to compensate for an undesired convolution by recreating the image as it existed before the convolution took place. This usually can be accomplished if the kernel for the convolution is known. If the kernel is not known then we have a much more difficult problem that has no general solution.

In general, the objective of deconvolution is to find f given k and g using the convolution equation

$$\mathbf{f} * \mathbf{k} = \mathbf{g}$$

Since the kernel can sometimes be singular, the inverse cannot be taken. To avoid this, one of the methods to perform deconvolution is by computing the Fourier Transform of the filtered image g and the kernel k. Our equation would look like,

$$\mathbf{F} = \mathbf{G}/\mathbf{K}$$

where F, K, and G are the Fourier Transforms of f, k, and g respectively. In the last step, the inverse Fourier Transform of G/K would be taken to find the deconvolved image f.

```
def deconvolve_fft(image, kernel):
    # Apply fft on image and kernel
    image_fft = np.fft.fftn(image)
    kernel_fft = np.fft.fftn(kernel)

    # Deconvolve
    matrix_fft = np.fft.ifftn(image_fft/kernel_fft)
```

```
# Rescale deconvolved image
deconvolve_img = matrix_fft/np.amax(matrix_fft)
return deconvolved_img.real
```

For the purposes of this lab, we know the kernel of the convolution by making a synthetic blur. Synthetic blur and real blur are similar, the only difference is the problem with noise in real blur. In most applications, noise is inevitable, and a real observation is often given by:

$$(u * k) + \eta = u_0$$

Here  $u$  is the original image,  $k$  is the kernel, and  $u_0$  is the blurred image with noise. If we assume that a noisy image has no noise, then when it is deconvolved we could lose information.

**Problem 4.** Run the `convolve_fft` and the `deconvolve_fft` using the `balloons_resized_bw.jpg` and the kernel from `motion_psf`. Then run the same functions again using the `add_noise` function below. Plot both resulting images.

Hint: Since the values may shift after being computed by `np.fft.ifftn`, rescale the deconvolved image.

```
def add_noise(image,n=300):
    """
    Adds salt and pepper noise to the image
    Parameters:
    -----
    image: ndarray of shape (n, m)
           numpy array of the image to be blurred
    n: int,
       controls how much noise is added
       A larger n indicates less noise
    Returns:
    -----
    noisy_image: ndarray of shape (n, m),
                numpy array representing the image with salt
                and pepper noise added
    """
    noisy_image = image.copy()

    noise_vec = [(np.random.randint(0,image.shape[0]), np.random.randint(0,
image.shape[1])) for k in xrange(image.shape[0] * image.shape[1] / n) ]

    for noise in noise_vec:
        noisy_image[noise] = np.random.rand()

    return noisy_image
```

Compare the images. Why shouldn't we assume that an observed image is noiseless?

## The Variational Model

Now the problem is this: How can we deblur a noisy and blurry image? Call the noisy blurred imaged  $u_0$  and consider the cost functional

$$\begin{aligned} J[u|u_0, k] &= \alpha \int_{\Omega} |\nabla u| + \frac{\lambda}{2} \int_{\Omega} (K[u] - u_0)^2 dx \\ &= \int_{\Omega} \left( \alpha |\nabla u| + \frac{\lambda}{2} (K[u] - u_0)^2 \right) dx \\ &= \int_{\Omega} L(u, \nabla u) dx \end{aligned}$$

where  $x = (x_1, x_2)$  is the position of a pixel,  $\Omega = \mathbb{R}^2$ ,  $\alpha$  and  $\lambda$  are positive weights, and  $K[u] = k * u$ . This penalizes both total variation in the image, as well as differences between a blurred version of the image and the noisy blurred input image. This can be minimized computationally using the same method in the Total Variation lab. The minimizer must satisfy the Euler-Lagrange equation:

$$L_u - \operatorname{div} L_{\nabla u} = 0$$

So we search for a steady-state solution to the parabolic PDE

$$u_t = -(L_u - \operatorname{div}(L_{\nabla u}))$$

which evaluates to

$$\begin{aligned} u_t &= \alpha \nabla \cdot \left[ \frac{\nabla u}{|\nabla u|} \right] - \lambda K^*[K[u] - u_0] \\ &= -\lambda K^*[K[u] - u_0] + \frac{u_{xx}u_y^2 + u_{yy}u_x^2 - 2u_xu_yu_{xy}}{(u_x^2 + u_y^2)^{3/2}} \end{aligned}$$

where  $K^*$  is the adjoint of  $K$  and  $u_{xx}$  and  $u_{yy}$  are approximated as

$$u_{xx} \approx \frac{u_{i+1,j}^n - 2u_{ij}^n + u_{i-1,j}^n}{\Delta x^2}, \quad (1.1)$$

$$u_{yy} \approx \frac{u_{i,j+1}^n - 2u_{ij}^n + u_{i,j-1}^n}{\Delta y^2}. \quad (1.2)$$

In order to calculate  $K^*$  numerically, we need a formula for it. Consider the following property of the adjoint:

$$\langle u, K[v] \rangle = \langle K^*[u], v \rangle$$

Now observe

$$\begin{aligned} \langle K^*[u], v \rangle &= \langle k * \bar{u}, \bar{v} \rangle \\ &= \int \left( \int \bar{k}(x, y) u(x) dx \right) \bar{v}(y) dy \\ &= \int \left( \int \overline{\bar{k}(x, y) v(y)} dy \right) u(x) dx \\ &= -\langle \bar{k} * u, v \rangle \end{aligned}$$

Thus if  $K[u] = k * u$ ,  $K^*[u] = -\bar{k} * u$ .

Also notice the singularity that occurs in the flow when  $|\nabla u| = 0$ . Numerically we will replace  $|\nabla u|^3$  in the denominator with  $(\epsilon + |\nabla u|^2)^{3/2}$ , to remove the singularity.



### 1.0.1 Numerical Implementation

```
def variational_calculus(f, k):
    """
    Computes the right hand side of our parabolic PDE
    Parameters:
    -----
    f - ndarray of shape (n, m)
        original noisy blurred image
    k - ndarray of shape (d, d)
        numpy array of a blurred kernel
    Returns:
    -----
    u1 - ndarray of shape(n, m)
        deblurred and denoised image
    """
    time_steps =
    delta_t =
    eps =
    lam =
    alpha =

    K_padded = np.zeros_like(f).astype('float32')
    K_padded[:k.shape[0], :k.shape[1]] = k[::-1,::-1]

    k_pad = np.zeros_like(f).astype('float32')
    k_pad[:k.shape[0], :k.shape[1]] = k

    fft_k_padded =

    def rhs(u):
        """
        Computes the right hand side of our parabolic PDE
        Parameters:
        -----
        u - ndarray of shape (n, m)
            copy/ iteration of the u_0
        Modifies:
        -----
        Euler - Lagrange of the variational deblurred model
        """
        ux =
        uy =

        uxx =
        uyy =

        uxy =
```

```

    var_deblurred =

    K_u = convolve2(u, k_pad)

    kstar = ifftn(fftn(K_u - u_0) * fft_k_padded).real

    u -= delta_t*(lam*kstar - alpha * var_deblurred)
    u0 = f.copy()
    u1 = f.copy()
    iteration = 0
    while iteration < time_steps:
        rhs(u1, k_pad)
        if la.norm(np.abs((u0 - u1))) < 1e-5:
            break
        u0 = u1.copy()
        iteration+=1

    return u1

```

**Problem 5.** Using  $\Delta t = 1e-3$ ,  $\lambda = .9$ ,  $\alpha = .6$ , implement the numerical scheme mentioned above to obtain the variational deblurring model and its Euler Lagrange equation. Take 200 steps in time. Your resulting image should not be noisy or blurry. How small should  $\epsilon$  be? Hint: To compute the spatial derivatives, consider the following:

```

u_x = (np.roll(u,-1,axis=1) - np.roll(u,1,axis=1))/2
u_xx = np.roll(u,-1,axis=1) - 2*u + np.roll(u,1,axis=1)
u_xy = (np.roll(u_x,-1,axis=0) - np.roll(u_x,1,axis=0))/2.

```