Gradient Descent and FISTA

Gradient descent is an iterative algorithm that finds the minimum of a convex function by following the slope "downhill" until it reaches a minimum. To solve the minimization problem

$$\text{minimize } g(\mathbf{x}),$$

we find the gradient of g wrt $\mathbf{x}$, $\nabla_{\mathbf{x}} g$, and use the property that the gradient always points in the direction of steepest *ascent*. In order to minimize g, we go the other direction:

$$\mathbf{x}_0 = \text{ initial guess}$$
$$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \alpha_k \nabla g(\mathbf{x}_k),$$

where α is a step size that determines how far in the descent direction we go at each iteration.

Applied to our problem:

$$g(\mathbf{v}) = \frac{1}{2}\|\mathbf{A}\mathbf{v} - \mathbf{b}\|_2^2$$
$$\nabla_{\mathbf{v}} g(\mathbf{v}) = \mathbf{A}^H(\mathbf{A}\mathbf{v} - \mathbf{b}),$$

where $\mathbf{A}^H$ is the adjoint of $\mathbf{A}$, $\mathbf{b}$ is the sensor measurement and $\mathbf{v}$ is the image of the scene.

We use more efficient variants of this algorithm, like Nesterov Momentum and FISTA, both of which are shown below.

Loading and preparing our images

In [1]:

```
import numpy as np
import numpy.fft as fft
import matplotlib.pyplot as plt
from IPython import display
from PIL import Image

%matplotlib inline
```

The code takes in two grayscale images: a point spread function (PSF) `psfname` and a sensor measurement `imgname`. The images can be downsampled by a factor f, which must be a of the form $1/2^k$, for some non negative integer k (typically between 1/2 and 1/8).

In [2]:

```
psfname = "./psf_sample.tif"
imgname = "./rawdata_hand_sample.tif"

# Downsampling factor (used to shrink images)
f = 1/8

# Number of iterations
iters = 100
```

In [3]:

```
def loaddata(show_im=True):
    psf = Image.open(psfname)
    psf = np.array(psf, dtype='float32')
    data = Image.open(imgname)
    data = np.array(data, dtype='float32')

    """In the picamera, there is a non-trivial background
    (even in the dark) that must be subtracted"""
    bg = np.mean(psf[5:15,5:15])
    psf -= bg
    data -= bg

    """Resize to a more manageable size to do reconstruction on.
    Because resizing is downsampling, it is subject to aliasing
    (artifacts produced by the periodic nature of sampling). Demosaicing is an attempt
    to account for/reduce the aliasing caused. In this application, we do the simplest
    possible demosaicing algorithm: smoothing/blurring the image with a box filter"""

    def resize(img, factor):
        num = int(-np.log2(factor))
        for i in range(num):
            img = 0.25*(img[::2,::2,...]+img[1::2,::2,...]+img[::2,1::2,...]+img[1::2,1::2,...])
        return img

    psf = resize(psf, f)
    data = resize(data, f)


    """Now we normalize the images so they have the same total power. Technically not a
    necessary step, but the optimal hyperparameters are a function of the total power in
    the PSF (among other things), so it makes sense to standardize it"""

    psf /= np.linalg.norm(psf.ravel())
    data /= np.linalg.norm(data.ravel())

    if show_im:
        fig1 = plt.figure()
        plt.imshow(psf, cmap='gray')
        plt.title('PSF')
        display.display(fig1)
        fig2 = plt.figure()
        plt.imshow(data, cmap='gray')
        plt.title('Raw data')
        display.display(fig2)
    return psf, data
```

Calculating convolutions using `fft`

We want to calculate convolutions efficiently. To do this, we use the "fast fourier transform" `fft2` which computes the Discrete Fourier Transform (DFT). The convolution theorem for DFTs only holds for circular convolutions. We can still recover a linear convolution by first padding the input images then cropping the output of the inverse DFT:

$$h * x = \mathcal{F}^{-1}[\mathcal{F}[h] \cdot \mathcal{F}[x]] = \mathrm{crop}\left[\, \mathrm{DFT}^{-1}\left\{\, \mathrm{DFT}[\,\mathrm{pad}[h]\,] \cdot \mathrm{DFT}[\,\mathrm{pad}[x]\,]\,\right\}\,\right]$$

Recovering the linear convolution correctly requires that we double the dimensions of our images. To take full advantage of the speed of the `fft2` algorithm, we actually pad `full-size`, which is the nearest power of two that is larger than that size.

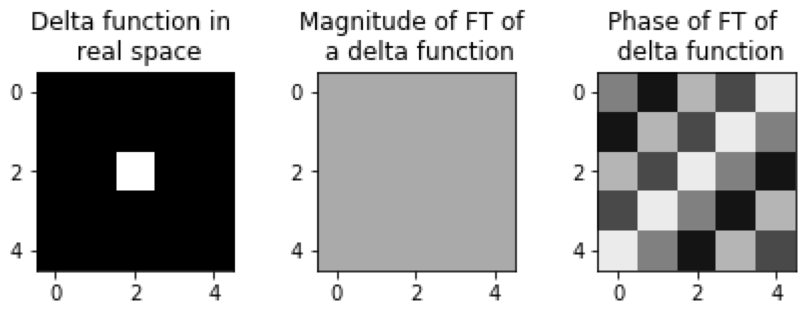
We have chosen `full-size` in such a way that it provides enough padding to make circular and linear convolutions look the same *after being cropped back down to* `sensor-size`. That way, the "sensor crop" due to the sensor's finite size and the "fft crop" above are the same, and we just need one crop function.

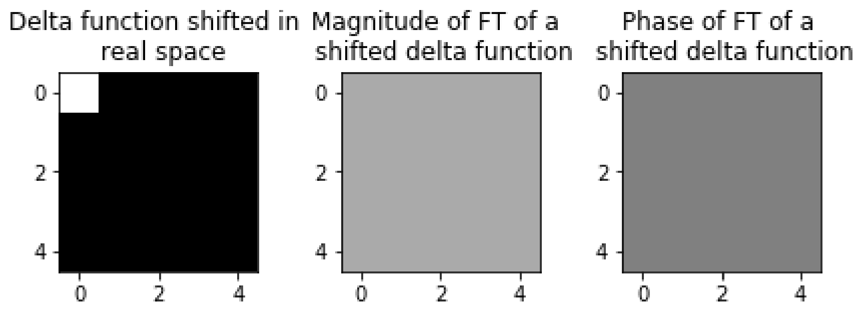
Along with initialization, we compute $\mathtt{H} = \mathtt{fft2}(\mathtt{hpad})$ and $\mathtt{Hadj} = \mathtt{H}^*$, which are constant matrices that will be needed to calculate the action of $\mathbf{A}$ and $\mathbf{A}^H$ at every iteration.

Lastly, we must take into account one more practical difference. In imaging, we often treat the center of the image as the origin of the coordinate system. This is theoretically convenient, but fft algorithms assume the origin of the image is the top left pixel. The magnitude of the fft doesn't change because of this distinction, but the phase does, since it is sensitive to shifts in real space. An example with the simplest function, a delta function, is displayed below. In order to correct this problem, we use `ifftshift` to move the origin of an image to the top left corner and `fftshift` to move the origin from the top left corner to the center.

In [4]:

```
def no_shift():
    delta = np.zeros((5,5))
    delta[2][2] = 1
    fft_mag = np.abs(fft.fft2(delta))
    fft_arg = np.angle(fft.fft2(delta))

    fig, ax = plt.subplots(nrows=1, ncols=3)
    fig.tight_layout()
    ax[0].imshow(delta, cmap='gray')
    ax[0].set_title('Delta function in \n real space')

    ax[1].imshow(fft_mag,vmin=-3,vmax=3,cmap='gray')
    ax[1].set_title('Magnitude of FT of \n a delta function')

    ax[2].imshow(fft_arg,vmin=-3,vmax=3,cmap='gray')
    ax[2].set_title('Phase of FT of \n delta function')

no_shift()

def shift():
    delta = np.zeros((5,5))
    delta[2][2] = 1
    delta_shifted = fft.ifftshift(delta)
    fft_mag = np.abs(fft.fft2(delta_shifted))
    fft_arg = np.angle(fft.fft2(delta_shifted))

    fig2, ax2 = plt.subplots(nrows=1, ncols=3)
    fig2.tight_layout()
    ax2[0].imshow(delta_shifted, cmap='gray')
    ax2[0].set_title('Delta function shifted in \n real space')

    ax2[1].imshow(fft_mag,vmin=-3,vmax=3,cmap='gray')
    ax2[1].set_title('Magnitude of FT of a \n shifted delta function')

    ax2[2].imshow(fft_arg,vmin=-3,vmax=3,cmap='gray')
    ax2[2].set_title('Phase of FT of a \n shifted delta function')

shift()
```





For this notebook and the ADMM notebook, we follow the following convention so we don't have to worry about this issue again:

1. All images in *real* space are stored with the origin in the center (so they can be displayed correctly)
2. All images in *Fourier* space are stored with the origin in the top left corner (so they can be used for processing correctly)
3. The above rules mean that, to perform a convolution between two real space images h and x, we do
$$\texttt{fftshift}(\texttt{ifft}[\texttt{fft}[\texttt{ifftshift}(h) \cdot \texttt{ifftshift}(x)]])$$
instead of
$$\texttt{ifft}[\texttt{fft}[h \cdot x]]$$
The rules imply that if we store the fourier transform of h for future use, instead of storing $\texttt{fft}[h]$, we store $\texttt{fft}[\texttt{ifftshift}(h)]$.

In [5]:

```
def initMatrices(h):
    pixel_start = (np.max(h) + np.min(h))/2
    x = np.ones(h.shape)*pixel_start

    init_shape = h.shape
    padded_shape = [nextPow2(2*n - 1) for n in init_shape]
    starti = (padded_shape[0]- init_shape[0])//2
    endi = starti + init_shape[0]
    startj = (padded_shape[1]//2) - (init_shape[1]//2)
    endj = startj + init_shape[1]
    hpad = np.zeros(padded_shape)
    hpad[starti:endi, startj:endj] = h

    H = fft.fft2(fft.ifftshift(hpad), norm="ortho")
    Hadj = np.conj(H)

    def crop(X):
        return X[starti:endi, startj:endj]

    def pad(v):
        vpad = np.zeros(padded_shape).astype(np.complex64)
        vpad[starti:endi, startj:endj] = v
        return vpad

    utils = [crop, pad]
    v = np.real(pad(x))

    return H, Hadj, v, utils

def nextPow2(n):
    return int(2**np.ceil(np.log2(n)))
```

Computing the gradient

The most important step in Gradient Descent is calculating the gradient
$$\nabla_{\mathbf{v}}\, g(\mathbf{v}) = \mathbf{A}^H(\mathbf{A}\mathbf{v} - \mathbf{b})$$
We do this in 2 steps:

1. We compute the action of $\mathbf{A}$ on $\mathbf{v}$, using `calcA`
2. We compute the action of $\mathbf{A}^H$ on $\texttt{diff} = \texttt{Av-b}$ using `calcAHerm`

Here, `vk` is the current padded estimate of the scene and `b` is the sensor measurement.

In [6]:

```python
def grad(Hadj, H, vk, b, crop, pad):
    Av = calcA(H, vk, crop)
    diff = Av - b
    return np.real(calcAHerm(Hadj, diff, pad))
```

We write $\mathbf{A}$ as:

$$\mathbf{Av} \iff \text{crop}\left[\mathcal{F}^{-1}\left\{\mathcal{F}(h)\cdot\mathcal{F}(v)\right\}\right]$$

In code, this becomes

$$\begin{aligned}\texttt{calcA(vk)} &= \texttt{crop}\left(\texttt{ifft}\left(\texttt{fft(hpad)}\cdot\texttt{fft(vk)}\right)\right)\\ &= \texttt{crop}\left(\texttt{ifft}\left(\texttt{H}\cdot\texttt{Vk}\right)\right)\end{aligned}$$

where $\cdot$ represents point-wise multiplication

In [7]:

```python
def calcA(H, vk, crop):
    Vk = fft.fft2(fft.ifftshift(vk))
    return crop(fft.fftshift(fft.ifft2(H*Vk)))
```

We first pad `diff` , giving us $\texttt{xpad}$, then we take the 2D fourier transform, $\texttt{X} = \mathcal{F}(\texttt{xpad})$. The action of the adjoint of A is

$$A^H\mathbf{x} \iff \mathcal{F}^{-1}\left\{\mathcal{F}(\mathbf{h})^*\cdot\mathcal{F}(\text{pad}[x])\right\}$$

This becomes

$$\begin{aligned}\texttt{calcAHerm(xk)} &= \texttt{ifft}\left(\left(\texttt{fft(h)}\right)^H\cdot\texttt{fft}\left(\texttt{pad(diff)}\right)\right)\\ &= \texttt{ifft}\left(\texttt{Hadj}\cdot\texttt{X}\right)\end{aligned}$$

In [8]:

```python
def calcAHerm(Hadj, diff, pad):
    xpad = pad(diff)
    X = fft.fft2(fft.ifftshift(xpad))
    return fft.fftshift(fft.ifft2(Hadj*X))
```

Putting it all together

This is the main function, which calculates the gradients and updates our estimation of the scene:

$$\begin{aligned}&\mathbf{v}_0 = \text{ anything}\\ &\text{for } k = 0 \text{ to num\_iters:}\\ &\qquad \mathbf{v}_{k+1} \leftarrow \texttt{gradient-update}(\mathbf{v}_k)\end{aligned}$$

There are different ways of doing the gradient update. The three we will show are regular GD, Nesterov momentum update, and FISTA.

To guarantee convergence, we set the step size to be

$$\alpha_k < \frac{2}{\|\mathbf{A}^H\mathbf{A}\|_2} \approx \frac{2}{\lambda_{max}(\mathbf{M}^H\mathbf{M})}$$

To calculate this, we use the property that $\mathbf{M}$ is diagonalizable by a Fourier Transform:

$$\begin{aligned}\mathbf{M}^H\mathbf{M} &= \left(\mathbf{F}^{-1}\text{diag}(\mathbf{Fh})\,\mathbf{F}\right)^H\,\mathbf{F}^{-1}\text{diag}(\mathbf{Fh})\,\mathbf{F}\\ &= \mathbf{F}^{-1}\text{diag}(\mathbf{Fh})^*\,\text{diag}(\mathbf{Fh})\,\mathbf{F}\\ \lambda_{max}(\mathbf{M}^H\mathbf{M}) &= \max\left(\text{diag}(\mathbf{Fh})^*\,\text{diag}(\mathbf{Fh})\right)\end{aligned}$$

In code, we have

$$\alpha = \frac{1.8}{\texttt{max}\,(\texttt{Hadj}\cdot\texttt{H})}$$

Since we are dealing with images, one constraint on the reconstructed image $\mathbf{v}_k$ is that all the entries have to be non-negative. We do this by doing projected gradient descent. The projection function `proj` we use is non-negativity, which projects `vk` onto the non-negative halfspace.

In [9]:

```
def grad_descent(h, b):
    H, Hadj, v, utils = initMatrices(h)
    crop = utils[0]
    pad = utils[1]

    alpha = np.real(1.8/(np.max(Hadj * H)))
    iterations = 0

    def non_neg(xi):
        xi = np.maximum(xi,0)
        return xi

    #proj = lambda x:x #Do no projection
    proj = non_neg #Enforce nonnegativity at every gradient step. Comment out as needed.


    parent_var = [H, Hadj, b, crop, pad, alpha, proj]

    vk = v



    #### uncomment for Nesterov momentum update ####
    #p = 0
    #mu = 0.9
    ################################################



    #### uncomment for FISTA update ################
    tk = 1
    xk = v
    ################################################

    for iterations in range(iters):

        # uncomment for regular GD update
        #vk = gd_update(vk, parent_var)

        # uncomment for Nesterov momentum update
        #vk, p = nesterov_update(vk, p, mu, parent_var)

        # uncomment for FISTA update
        vk, tk, xk = fista_update(vk, tk, xk, parent_var)

        if iterations % 10 == 0:
            image = proj(crop(vk))
            f = plt.figure(1)
            plt.imshow(image, cmap='gray')
            plt.title('Reconstruction after iteration {}'.format(iterations))
            display.display(f)
            display.clear_output(wait=True)


    return proj(crop(vk))
```

Gradient descent algorithms

Regular Gradient Descent

Regular gradient descent is simply following the negative of the gradient until we reach the minimum:

$$\begin{aligned}&\texttt{gradient-update}(\mathbf{v}_k):\\ &\qquad \mathbf{v}'_{k+1} \leftarrow \mathbf{v}_k - \alpha_k \mathbf{A}^H(\mathbf{A}\mathbf{v}_k - \mathbf{b})\\ &\qquad \mathbf{v}_{k+1} \leftarrow \mathrm{proj}_{\mathbf{v}\geq 0}(\mathbf{v}'_{k+1})\end{aligned}$$

In [10]:

```
def gd_update(vk, parent_var):
    H, Hadj, b, crop, pad, alpha, proj = parent_var

    gradient = grad(Hadj, H, vk, b, crop, pad)
    vk -= alpha*gradient
    vk = proj(vk)

    return xk
```

Nesterov Momentum

GD works but it's slow. One way to speed it up is to consider a velocity term, $\mathbf{p}$. Each update becomes:

$$\begin{aligned}&\texttt{gradient-update}(\mathbf{v}_k):\\ &\qquad \mathbf{p}_{k+1} \leftarrow \mu\mathbf{p}_k - \alpha_k \texttt{grad}(\mathbf{v}_k)\\ &\qquad \mathbf{v}'_{k+1} \leftarrow \mathbf{v}_k - \mu\mathbf{p}_k + (1+\mu)\mathbf{p}_{k+1}\\ &\qquad \mathbf{v}_{k+1} \leftarrow \mathrm{proj}_{\mathbf{v}\geq 0}(\mathbf{v}'_{k+1})\end{aligned}$$

The parameter μ is called the momentum and is strictly between 0 and 1.

In [11]:

```
def nesterov_update(vk, p, mu, parent_var):
    H, Hadj, b, crop, pad, alpha, proj = parent_var

    p_prev = p
    gradient = grad(Hadj, H, vk, b, crop, pad)
    p = mu*p - alpha*gradient
    vk += -mu*p_prev + (1+mu)*p
    vk = proj(vk)

    return vk, p
```

FISTA

Instead of Nesterov momentum, we can use FISTA, which speeds up the iterative process. Each iteration of the algorithm is as follows:

```
gradient-update($\mathbf{v}_k$):
    $\mathbf{v}_k \leftarrow \mathbf{v}_k - \alpha_k \texttt{grad}(\mathbf{v}_k)$
    $x_k \leftarrow \texttt{proj}(\mathbf{v}_k)$
    $t_{k+1} \leftarrow \frac{1+\sqrt{1+4t_k^2}}{2}$
    $\mathbf{v}_{k+1} \leftarrow x_k + \frac{t_k - 1}{t_{k+1}}(x_k - x_{k-1})$
```

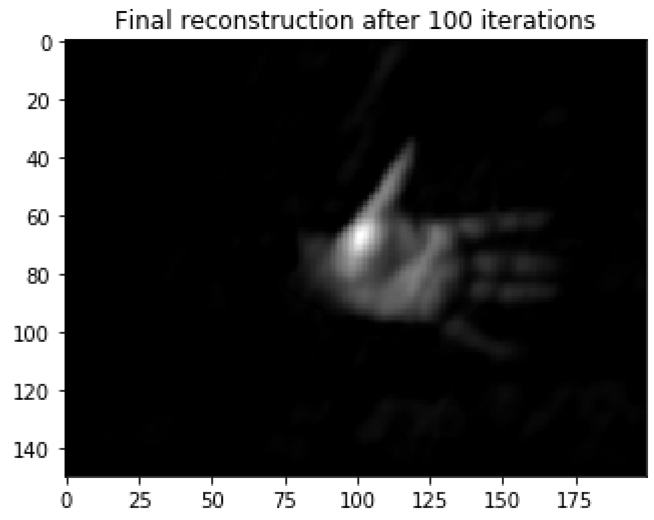
In [12]:

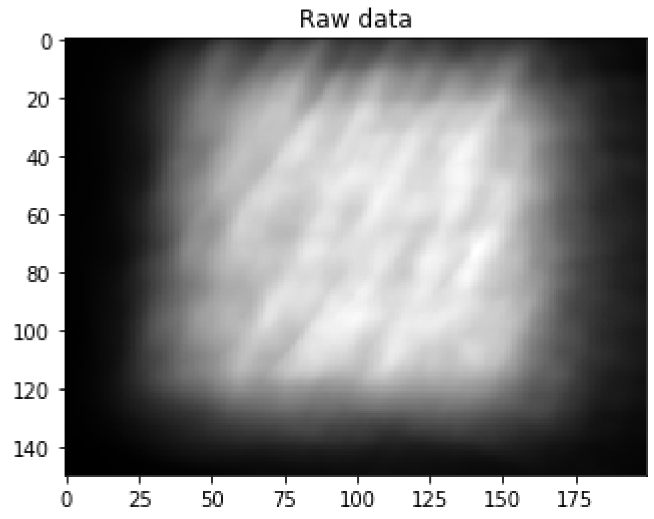
```
def fista_update(vk, tk, xk, parent_var):
    H, Hadj, b, crop, pad, alpha, proj = parent_var

    x_k1 = xk
    gradient = grad(Hadj, H, vk, b, crop, pad)
    vk -= alpha*gradient
    xk = proj(vk)
    t_k1 = (1+np.sqrt(1+4*tk**2))/2
    vk = xk+(tk-1)/t_k1*(xk - x_k1)
    tk = t_k1

    return vk, tk, xk
```

Running the algorithm

In [14]:

```
psf, data = loaddata()
final_im = grad_descent(psf, data)
plt.imshow(final_im, cmap='gray')
plt.title('Final reconstruction after {} iterations'.format(iters))
display.display()
```





In [ ]:

```
```