**CSE 559A/Fall 2020. Problem Set 1 Solution Key.**

*It is a violation of the academic integrity policy to scan, copy, or otherwise share these solutions*

1.(a) Simply add up the variance of the two noise terms (shot noise and additive noise) from the slides: $g^2 I_0 + g^2\sigma_{2a}^2 + \sigma_{2b}^2$.

(b) Replacing $g$ with $kg$ and $I_0$ with $I_0/k$ in the answer of (a), $kg^2 I_0 + k^2 g^2 \sigma_{2a}^2 + \sigma_{2b}^2$,

(c) The noise variables $k$ different samples are independent. Let the noise in each sample be $\epsilon_i$ which all have the same variance. Then we have, noise in the average $\epsilon = \frac{1}{k}\sum_{i=1}^{k}\epsilon_i$, which has variance $\frac{1}{k^2}\sum_{i=1}^{k}\mathrm{Var}(\epsilon_i) = \frac{1}{k}\mathrm{Var}(\epsilon_0)$. So averaging over k samples has the effect of producing an estimate with noise variance that is $1/k$ of the individual sample variances.

So we have, noise variance in the average $g^2 I_0 + kg^2\sigma_{2a}^2 + \frac{1}{k}\sigma_{2b}^2$.

(d) It depends! Taking $k$ shorter samples and averaging has no effect on the shot noise (expected, since the total number of photons is the same), scales up the pre-amplifier noise up, and the post-amplifier noise down, both by a factor of $k$. So this strategy is beneficial when $\sigma_{2b}^2$ is higher than $g^2\sigma_{2a}^2$.

2. Here is the code of the histogram equalization function.

```
  def histeq(X):
    ret = np.unique(X,return_inverse=True,return_counts=True)

    cdf = np.cumsum(np.float32(ret[2]))
    cdf = cdf - cdf[0]
    cdf = cdf / cdf[-1]
    cdf = np.round(cdf * 255)

    Y = cdf[ret[1]]
    return np.reshape(Y,X.shape)
```

3. Here is the code for magnitude computation and non-maxima suppression.

```
# Return magnitude, theta of gradients of X
def grads(X):
    df = np.float32([[1,0,-1]])
    sf = np.float32([[1,2,1]])

    gx = conv2(X,sf.T,'same','symm')
    gx = conv2(gx,df,'same','symm')

    gy = conv2(X,sf,'same','symm')
    gy = conv2(gy,df.T,'same','symm')

    H = np.sqrt(gx*gx+gy*gy)
    theta = np.arctan2(gy,gx)

    return H,theta
```

```
def nms(E,H,theta):
    E2 = E > 0
    ht, wd = E.shape[0], E.shape[1]
    theta = np.mod(np.round(theta*4.0/np.pi), 4)

    # Horizontal
    yx = np.where(np.logical_and(E2,theta == 0))
    E2[yx[0],yx[1]] = np.logical_and(
        H[yx[0],yx[1]] > H[yx[0],np.minimum(wd-1,yx[1]+1)],
        H[yx[0],yx[1]] > H[yx[0],np.maximum(0,yx[1]-1)])
```

```
    # Vertical
    yx = np.where(np.logical_and(E2,theta == 2))
    E2[yx[0],yx[1]] = np.logical_and(
        H[yx[0],yx[1]] > H[np.minimum(ht-1,yx[0]+1),yx[1]],
        H[yx[0],yx[1]] > H[np.maximum(0,yx[0]-1),yx[1]])

    # Diagonal
    yx = np.where(np.logical_and(E2,theta == 1))
    E2[yx[0],yx[1]] = np.logical_and(
        H[yx[0],yx[1]] > H[np.minimum(ht-1,yx[0]+1),np.minimum(wd-1,yx[1]+1)],
        H[yx[0],yx[1]] > H[np.maximum(0,yx[0]-1),yx[1]-1])

    # Anti-diagonal
    yx = np.where(np.logical_and(E2,theta == 3))
    E2[yx[0],yx[1]] = np.logical_and(
        H[yx[0],yx[1]] > H[np.maximum(0,yx[0]-1),np.minimum(wd-1,yx[1]+1)],
        H[yx[0],yx[1]] > H[np.minimum(ht-1,yx[0]+1),yx[1]-1])

    return np.float32(E2)
```

NMS should create results that look like this:

Without NMS

With NMS

Notice how diagonal edges are thinned.

4. Here is the code for bilateral filtering.

```
def bfilt(X,K,sgm_s,sgm_i):
    H = X.shape[0]; W = X.shape[1]

    yy = np.zeros(X.shape)
    B = np.zeros([H,W,1])


    for y in range(-K,K+1):
        for x in range(-K,K+1):
            if y < 0:
                y1a = 0; y1b = -y; y2a = H+y; y2b = H
            else:
                y1a = y; y1b = 0; y2a = H; y2b = H-y
            if x < 0:
                x1a = 0; x1b = -x; x2a = W+x; x2b = W
            else:
                x1a = x; x1b = 0; x2a = W; x2b = W-x


            bxy = X[y1a:y2a,x1a:x2a,:] - X[y1b:y2b,x1b:x2b,:]
            bxy = np.sum(bxy*bxy,axis=2,keepdims=True)
```

```
            bxy = bxy/(sgm_i**2) + np.float32( y**2 + x**2)/(sgm_s**2)
            bxy = np.exp(-bxy/2.0)

            B[y1b:y2b,x1b:x2b,:] = B[y1b:y2b,x1b:x2b,:]+bxy
            yy[y1b:y2b,x1b:x2b,:] = yy[y1b:y2b,x1b:x2b,:]+bxy*X[y1a:y2a,x1a:x2a,:]

    return yy/B
```

5. (a) Let's assume width is odd. You can keep $F[u,:]$ for $u \in \{0,1,\ldots(W-1)/2\}$ and get the values of $F[u,:]$ for the remaining $u$ from the fact that $F[W-u,H-v] = \bar{F}[u,v]$. But you're still storing a little more than half the locations. So, take $F[0,:]$ and keep only half the $v$ locations because you can get the rest as $F[0,v] = \bar{F}[0,H-v]$. If the height is odd, you can save $(H-1)/2$ of $F[0,:]$, plus drop the complex part of $F[0,0]$ since it is real. If the height is even, you will need to store $H/2+1$ locations as both $F[0,0]$ and $F[0,H/2]$ need to be stored, but both are real (in the Fourier equation, its $\exp(-jn_y\pi)$) so you store one number for each.

If width is even but the height is odd, you can apply the above idea to height then width.

If both height and width are even, let's say we split on width. Keep $F[u,:]$ for $u \in \{0,1,\ldots,W/2\}$. So that is $H*(W/2+1)$ complex numbers, which means $H$ extra complex numbers than we want to save. Consider the $H/2$ complex numbers for $F[0,:]$. Apply the same trick as above (keep half, $F[0,0]$ and $F[0,H/2]$ are both real). Moreover, you can apply the same trick—as for $F[0,:]$—to $F[W/2,:]$, since $F[W/2,v] = \bar{F}[W/2,H-v]$. This is because $\exp j(n\pi+\theta)$ and $\exp j(n\pi-\theta)$ are complex conjugates for both odd and even $n$. Similarly, both $F[W/2,0]$ and $F[W/2,H/2]$ are real and you need to store only one number for each.

(b) Here is the code for the padding function.

```
def kernpad(K,size):
    Ko = np.zeros(size,dtype=np.float32)
    ks = K.shape
    hk = (ks[0]-1)//2
    wk = (ks[1]-1)//2

    Ko[:(hk+1),:(wk+1)] = K[hk:,wk:]
    Ko[:(hk+1),-wk:] = K[hk:,:wk]
    Ko[-hk:,:(wk+1)] = K[:hk,wk:]
    Ko[-hk:,-wk:] = K[:hk,:wk]

    return Ko
```

6. Here is the code for wavelet decomposition and synthesis.

```
def im2wv(img,nLev):
    if nLev == 0:
        return [img]

    hA = (img[0::2,:] + img[1::2,:])/2.
    hB = (-img[0::2,:] + img[1::2,:])/2.

    L = hA[:,0::2]+hA[:,1::2]
    h1 = hB[:,0::2]+hB[:,1::2]
    h2 = -hA[:,0::2]+hA[:,1::2]
    h3 = -hB[:,0::2]+hB[:,1::2]

    return [[h1,h2,h3]] + im2wv(L,nLev-1)
```

```
def wv2im(pyr):
    while len(pyr) > 1:
        L0 = pyr[-1]
```

```
        Hs = pyr[-2]
        H1 = Hs[0]
        H2 = Hs[1]
        H3 = Hs[2]

        sz = L0.shape
        L = np.zeros([sz[0]*2,sz[1]*2],dtype=np.float32)

        L[::2,::2] = (L0-H1-H2+H3)/2.
        L[1::2,::2] = (L0+H1-H2-H3)/2.
        L[::2,1::2] = (L0-H1+H2-H3)/2.
        L[1::2,1::2] = (L0+H1+H2+H3)/2.

        pyr = pyr[:-2] + [L]

    return pyr[0]
```