Lecture 14

Image-based Operations as Matrix-Vector Operations

14.1 Lecture Objectives

- Become familiar with mapping two essential algebraic operations, matrix-vector product and matrix transpose-vector product, into computational implementations
- Get ready to make the leap from solving small toy problems to solving large imaging-scale problems

14.2 DFT

If we view the DFT operation as a matrix-vector operation, we can implement it by simply reshaping our vector into an image with the right dimensions, applying the DFT using an FFT algorithm, and reshaping the output back into a vector. In Matlab notation, this might look something like:

```
% y = Fx where:
% F is a matrix that implements a 2D DFT (with fftshifts)
% x is the vectorized version of the input image
% y is the vectorized version of the output DFT
x2 = reshape(x,[N,N]); % Assuming the image is size NxN
y2 = fftshift(fft2(ifftshift(x2)));
y = reshape(y2,[N*N,1]);
```

Now, the question is how to implement the transpose of this operation (Hermitian transpose in the case of DFT since it is a complex-valued operation). However, for the case of the DFT, we know that this is an orthogonal operation where its transpose is its inverse (up to a scaling factor given by the size of the array). In other words, the transpose of the DFT matrix is the iDFT matrix (up to some scaling factor since the DFT matrix is not normalized):

```
% y = F^H x where:
% F is a matrix that implements a 2D DFT (with fftshifts)
% x is the vectorized version of the input DFT domain array (of size NxN)
% y is the vectorized version of the output image-domain signal
x2 = reshape(x,[N,N]); % Assuming the image is size NxN
y2 = N^2*fftshift(ifft2(ifftshift(x2)));
y = reshape(y2,[N*N,1]);
```

14.3 Subsampling

In our reconstructions, we often want to fit data acquired at a subset of the Fourier locations provided by the DFT. For instance, we may skip some lines or some locations within an $N \times N$ Cartesian array. In this case, the forward matrix can be viewed as $\mathbf{A} = \mathbf{UF}$ where $\mathbf{F}$ performs a 2D DFT, and $\mathbf{U}$ is a diagonal matrix with all the rows corresponding to unsampled Fourier space entries removed.

Implementing the effect of $\mathbf{U}$ on a vectorized array is easy. Suppose we have a $N \times N$ array and we have a binary 'mask' (call it 'M') that specifies which entries of the array we are actually sampling:

```
% y = Ux where:
% U is a matrix that implements subsampling determined by a binary mask M
% x is the vectorized version of the input array
% y is the vectorized version of the output array
y = x(M(:));
% Note the use of the '(:)' notation to vectorize the binary array M
```

The transpose of this operation is the zero-padding operation, where we take the input (of length the number of sampled entries), and add zeroes at the non-sampled entries:

```
% y = U^T x where:
% U is a matrix that implements subsampling determined by a binary mask M
% x is the vectorized version of the input array
% y is the vectorized version of the output array
y = zeros(N*N,1);
y(M(:)) = x;
% Note the use of the '(:)' notation to vectorize the binary array M
```

14.4 Radon Transform

Given an image and a set of projection angles, we can perform a radon transform (ie: projections along a set of angles), as follows:

```
% y = Px where:
% P is a matrix that performs projections along an array of angles 'theta'
% x is the vectorized version of the input image
% y is the vectorized version of the output array
x2 = reshape(x,[N,N]); % Assuming the image is size NxN
y2 = radon(x2,theta);
y = reshape(y2,[],1);
```

Importantly, the transpose of the radon transform is the (unfiltered) backprojection (including some additional scaling, depending on implementation):

```
% y = P^H x where:
% P is a matrix that performs projections along an array of angles 'theta'
% x is the vectorized version of the projection data
% y is the vectorized version of the output array (image)
x2 = reshape(x,[],Nangle); % Assuming Nangle projection angles were used
y2 = c*iradon(x2,theta,'spline','none',1,N);
% Scaling by c = 2/pi*Nangle seems to work with Matlab's implementation of backprojec
y = reshape(y2,[],1);
```

14.5 High-Pass Filtering (Finite Differences)

Oftentimes we want to implement a high pass filter as part of a regularization term (ie: $\|\mathbf{D}\mathbf{x}\|_p$). Here are two alternative implementations of the filter and its transpose: i) based on image shifts and additions/subtractions, ii) based on creating a finite-difference taking matrix in 1D, and extending to 2D using the `kron` command. In this implementation, we keep the memory requirements in check by making our huge matrix `sparse` (a type of array allocation in Matlab, efficient for storing and operating with sparse arrays).

```
%% Filtering (Finite differences with periodic boundary condition)
% y = D x; (apply filtering)
% z = D^H y; (apply transpose of filtering)

N = 256;
x = phantom(N);

% Implementation 1
y = 4*x - circshift(x,[1, 0]) - circshift(x,[-1,  0]) - ...
    circshift(x,[0, 1]) - circshift(x,[0, -1]);
z = 4*y - circshift(y,[1, 0]) - circshift(y,[-1,  0]) - ...
    circshift(y,[0, 1]) - circshift(y,[0, -1]);
y = y(:);
z = z(:);
```

```
% Implementation 2
D = 2*eye(N) - circshift(eye(N),[0, -1]) - circshift(eye(N),[0, 1]);
D = sparse(D);
I = speye(N);
D2 = kron(I,D) + kron(D,I);
y2 = A2*x(:);
z2 = A2'*y2(:);
```