Homework #10, EECS 598-006, W20. Due **Thu. Apr. 03**, by 4:00PM

1. [41] **Image super-resolution using wavelet sparsity regularizer**

In a **image super-resolution** problem, we are given a low-resolution image $\boldsymbol{y} = \text{vec}(\boldsymbol{Y})$ and the goal is to create a higher resolution image $\boldsymbol{x} = \text{vec}(\boldsymbol{X})$ from it. Usually there is noise in the given image too, so an appropriate measurement model is $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{\varepsilon}$, where $\boldsymbol{A}$ is a matrix (linear map) representing the down sampling operation. If we believe that the higher resolution image has sparse wavelet transform coefficients, then a reasonable optimization problem is:

$$\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}} \Psi(\boldsymbol{x}), \quad \Psi(\boldsymbol{x}) = \frac{1}{2} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \beta \|\boldsymbol{D}\boldsymbol{W}\boldsymbol{x}\|_1,$$

where $\boldsymbol{W}$ denotes an orthogonal discrete wavelet transform, and $\boldsymbol{D}$ is a diagonal matrix of 0 and 1 values to select the wavelet detail coefficients.

(a) [10] To solve the above optimization problem, we need code for $\boldsymbol{A}$, which means we first need a mathematical model for how the low-resolution image $\boldsymbol{y}$ relates to the high-resolution image $\boldsymbol{x}$ in the absence of noise. If $x[m,n]$ is a $M \times N$ digital image for $m = 0, \ldots, M-1$ and $n = 0, \ldots, N-1$, where M and N are even, then a typical model for a factor of two down-sampling is

$$y[m,n] = \frac{1}{4}(x[2m,2n] + x[2m+1,2n] + x[2m,2n+1] + x[2m+1,2n+1]), \quad \begin{array}{l} m = 0, \ldots, M/2-1 \\ n = 0, \ldots, N/2-1. \end{array}$$

Study the following code that implements $\boldsymbol{A}$ as a `LinearMapAA` object.

```
using LinearMapsAA
(n1,n2) = (64,128) # test size (M,N) just for illustration
down1 = (x) -> (x[1:2:end,:] + x[2:2:end,:])/2 # 1D down-sampling by 2x
down2 = (x) -> down1(down1(x)')' # 2D down-sampling by factor of 2x
A = LinearMapAA(x -> down2(reshape(x,n1,n2))[:], (Int((n1/2)*(n2/2)), n1*n2))
```

The size of $\boldsymbol{A}$ is $(MN/4) \times (MN)$ which would be too large to store for realistic image sizes, so we use `LinearMapAA`. To use this $\boldsymbol{A}$ for optimization, you will also need a method for implementing the **adjoint** operation corresponding to multiplying by the transpose $\boldsymbol{A}'$. Think about the linear operation above and examine `Matrix(A)'` for small image sizes. Then write a subroutine that performs the adjoint operation efficiently. Do not use any `sparse` functions.

Hint. The general ideas here are similar to the earlier HW involving `diff2d_adj`.

Your file should be named `down2_adj.jl` and should contain the following function:

```
"""
    x = down2_adj(y)

Let `down2` denote the linear downsampling operation where each 2×2 block
of image pixels is averaged to form one output pixel.
This routine returns the *adjoint* of that linear operation.

in
- `y` `[n1 n2]` where `n1` and `n2` are even.

out
- `x` `[2*n1 2*n2]`

"""
function down2_adj(array::AbstractArray{<:Number,2})
```

Submit your solution to mailto:eecs556@autograder.eecs.umich.edu.

(b) [0] Use your subroutine as part of the second argument of the `LinearMapAA` call, *i.e.*, `y -> down2_adj ???`

Then test it for a small image size by the command: `Matrix(A)' == Matrix(A')`

Hint: think about `reshape` and `[:]` here.

(c) [3] Determine the Lipschitz constant of the gradient of the data term above. The answer is a number and you do not need `opnorm` to find it. Hint. First consider the case where the input image size is just 2×2.

(d) [10] Write a script that applies 10 iterations of POGM to minimize the cost function above for data generated as follows and produces the plots and images in the subsequent parts.

```
using Random: seed!
using LinearMapsAA, Plots
using MIRT: Aodwt, pogm_restart, jim, ellipse_im
nx,ny = 192,256
Xtrue = ellipse_im(ny, oversample=2)[Int((ny-nx)/2+1):Int(ny-(ny-nx)/2),:]
down1 = (x) -> (x[1:2:end,:] + x[2:2:end,:])/2 # 1D down-sampling by 2x
down2 = (x) -> down1(down1(x)')' # 2D down-sampling by factor of 2x
Ytrue = down2(Xtrue); seed!(0); sig=0.1; Y = Ytrue + sig * randn(size(Ytrue))
W,scales,mfun = Aodwt((nx,ny)) # orth. discrete wavelet transform (LinearMap)
plot(jim(Xtrue, "true"), jim(Ytrue, "lo-res"), jim(Y, "noisy"))
```

Use $\beta = 0.05$ here. Also, for the 1-norm above, only regularize the wavelet detail coefficients, not the wavelet approximation coefficients, just as you did in a previous HW problem.
Submit a screenshot of your code to gradescope.

(e) [5] To apply any iterative algorithm to that cost function, we need an initial image $\boldsymbol{x}_0 = \mathrm{vec}(\boldsymbol{X}_0)$. For this application, the initial image $\boldsymbol{x}_0$ should be computed from $\boldsymbol{y}$ by replicating each pixel in $\boldsymbol{y}$ twice in each direction.

For example, if $\boldsymbol{Y} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ then $\boldsymbol{X}_0 = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix}$, for which $\boldsymbol{y} = \mathrm{vec}(\boldsymbol{Y}) = \boldsymbol{A}\boldsymbol{x}_0 = \boldsymbol{A}\,\mathrm{vec}(\boldsymbol{X}_0)$.

Submit a screen shot of your initial image to gradescope. It should look pretty similar to the true image.

(f) [5] Plot the cost function versus iteration k for the POGM approach.
You should see that POGM converges quite quickly, probably because $\boldsymbol{W}$ is unitary and $\boldsymbol{A}'\boldsymbol{A}$ is block diagonal.

(g) [5] Show images of the true $\boldsymbol{x}$, the noisy low-resolution image $\boldsymbol{y}$, the initial image $\boldsymbol{x}_0$, and the final image $\hat{\boldsymbol{x}}$.

(h) [3] Report the NRMSE values of $\boldsymbol{x}_0$ and $\hat{\boldsymbol{x}}$.

(i) [0] You will find that the NRMSE improves only a little. Speculate why.

(j) [0] Optional. Explore other wavelet types `https://github.com/JuliaDSP/Wavelets.jl` using the optional argument of `Aodwt` to try to improve the results.

2. [26] **Compressed sensing MRI**

This problem focuses on a relatively simple version of image reconstruction for magnetic resonance imaging (MRI). A simple model for 2D MRI is that the data consists of samples of the 2D **DFT** of a 2D slice of the object being scanned.

If $\boldsymbol{X}$ denotes a $M \times N$ (discretized) slice of the patient, then the data model for "fully sampled" 2D MRI is

$$\boldsymbol{y} = \texttt{fft(X)[:]} + \boldsymbol{\varepsilon} \in \mathbb{C}^{MN}$$

where the JULIA `fft` function computes the 2D FFT of a 2D input argument, and $\boldsymbol{\varepsilon}$ denotes a complex additive Gaussian noise vector of length MN. If we collect such fully sampled measurements, then image reconstruction is a trivial inverse 2D FFT:

$$\hat{\boldsymbol{X}} = \texttt{ifft(reshape(y, M, N))} .$$

One way to reduce scan time in MRI is to collect fewer than MN samples for a $M \times N$ image and then used **compressed sensing** methods to estimate $\boldsymbol{X}$ from $\boldsymbol{y}$. Let `samp` denote a boolean $M \times N$ array that is `true` for DFT coefficients that we sample, and `false` otherwise and let $K =$ `sum(samp)` $\leq MN$ denote then number of samples. Then for such "under-sampled" scans the measurement model becomes:

$$\boldsymbol{y} = \texttt{fft(X)[samp]} + \boldsymbol{\varepsilon} \in \mathbb{C}^{K}.$$

Mathematically we can write this as

$$\boldsymbol{y} = \boldsymbol{F}\boldsymbol{x} + \boldsymbol{\varepsilon}$$

where $\boldsymbol{x} = \text{vec}(\boldsymbol{X})$ and $\boldsymbol{F}$ denotes the $K \times MN$ matrix consisting of the K rows of the DFT corresponding to the elements of `samp`. Two equivalent ways to make $\boldsymbol{F}$ in JULIA for a 1D signal are:

```
F = exp.(-2im*pi*(findall(samp).-1)*(0:N-1)'/N)
```

```
F = exp.(-2im*pi*(0:N-1)*(0:N-1)'/N)[samp,:]
```

Such code is incomplete for the 2D DFT, and uses too much memory for large problems anyway.
We must use something like a `LinearMapAA` to represent $\boldsymbol{F}$, *e.g.*, as follows:

```
F = LinearMapAA(x -> fft(reshape(x, M, N))[samp], (sum(samp), M*N) ; T=ComplexF32)
```

You should think carefully about all of the arguments used in the above `LinearMapAA` call!

A typical compressed sensing model is to assume that $\boldsymbol{T}\boldsymbol{x}$ is sparse for some transform $\boldsymbol{T}$, such as a wavelet transform. Under that model, a reasonable estimator is

$$\hat{\boldsymbol{x}} = \operatorname*{arg\,min}_{\boldsymbol{x} \in \mathbb{C}^{MN}} \frac{1}{2} \|\boldsymbol{F}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \beta \|\boldsymbol{D}\boldsymbol{T}\boldsymbol{x}\|_1 ,$$

where $\boldsymbol{D}$ is a diagonal weighting matrix. For now, we focus on the case where $\boldsymbol{T}$ is a unitary matrix, specifically an **orthogonal discrete wavelet transform**. As seen previously, the **proximal optimized gradient method** (**POGM**) is well-suited to such problems.

(a) [3] You are going to apply POGM to data generated as follows:

```
using Random: seed!
using FFTW: fft
using MIRT: Aodwt, jim
M,N = 192,256; Xtrue = zeros(M,N);
Xtrue[30:50,20:90] .= 1; Xtrue[90:100,100:110] .= 1; Xtrue[130:150,20:90] .= 1;
Xtrue[20:170,150:200] .= 1; Xtrue[150:151,160:161] .= 0
seed!(0); sampfrac = 0.3; samp = rand(M,N) .< sampfrac; sig = 1
mod2 = (N) -> mod.((0:N-1) .+ Int(N/2), N) .- Int(N/2)
samp .|= (abs.(mod2(M)) .< Int(M/8)) * (abs.(mod2(N)) .< Int(N/8))' # center
ytrue = fft(Xtrue)[samp]; y = ytrue + sig * randn(size(ytrue)) +
    1im * sig * randn(size(ytrue)); # complex noise!
T,scales,mfun = Aodwt((M,N)) # Orthogonal disc. wavelet transform (LinearMapAA)
```

As an easy warm-up, generate the data and then display the true image $\boldsymbol{X}_{\text{true}}$ and the sampling pattern as follows:
`plot(jim(Xtrue), jim(samp))`

Make an initial $M \times N$ image $\boldsymbol{X}_0$ by taking the inverse FFT of "zero-filled" k-space data, defined as follows:

```
zfill = zeros(eltype(y), M,N); zfill[samp] = y
```

Let `X0` denote the inverse FFT of that data.
Make a nice display of these initial ingredients:

```
plot(jim(Xtrue,"Xtrue"), jim(samp, "sampling", fft0=true), jim(X0,"X0"))
```

If your code is correct, $\boldsymbol{X}_0$ should look like a blurry version of $\boldsymbol{X}_{\text{true}}$ because it is missing many high spatial frequency components that correspond to fine details. (The `fft0=true` option displays the DFT coefficients with 0 in the middle, akin to MATLAB's `fftshift` command, which is usually more intuitive.)
Optional: also show the wavelet detail coefficients of $\boldsymbol{X}_{\text{true}}$.
Submit a screenshot of your figure to gradescope.

(b) [0] Can you explain the sampling pattern? If not, ask someone in class who knows about MRI.

(c) [3] To apply a gradient-based method, we need the (best) Lipschitz constant L for the data term above. Determine L.
Hint. $\boldsymbol{F} = \boldsymbol{P}\sqrt{MN}(\boldsymbol{Q}_N \otimes \boldsymbol{Q}_M)$, where $\boldsymbol{Q}_N$ having elements $Q_{kn} = \frac{1}{\sqrt{N}} \exp(-\imath 2\pi kn/N)$ denotes the $N \times N$ **unitary DFT** matrix, and $\boldsymbol{P}$ denotes the $K \times MN$ matrix that is all zeros except for a single 1 in each row that selects the DFT coefficients that we sample. Specifically: $\boldsymbol{P}\boldsymbol{x} =$ `x[samp]`. Now think about $\boldsymbol{P}'\boldsymbol{P}$.

(d) [5] The gradient of the data term above is $\boldsymbol{F}'(\boldsymbol{F}\boldsymbol{x} - \boldsymbol{y})$, so to apply any gradient-based method to this optimization problem, we need the adjoint operation $\boldsymbol{F}'$. Modify the initial `LinearMapAA` definition given above to provide that capability.
Hint. If $\boldsymbol{A}\,\mathsf{vec}(\boldsymbol{X}) =$ `fft(X)[:]`, then $\boldsymbol{A}$ is not unitary, but $\boldsymbol{A}^{-1} = \frac{1}{MN}\boldsymbol{A}'$. See **inverse DFT**.
Hint. MIRT.jl includes a function `embed` that may be useful.

Write a JULIA script that runs POGM and produces the figures below.
Submit a screenshot of your script, including the modified `LinearMapAA` call, to gradescope.
Choose the diagonal weighting matrix $\boldsymbol{D}$ to regularize only the detail wavelet coefficients.
Use $\beta = 0.004MN$ and 100 iterations.

(e) [5] Plot the cost function $\Psi(\boldsymbol{x}_k)$ (no logarithm) versus iteration k.
Optional: compare to ISTA and FISTA.

(f) [5] Plot the peak signal-to-noise ratio (PSNR) of $\boldsymbol{x}_k$ versus iteration k, where PSNR is defined by

$$20\log_{10}\left(\frac{\|\mathsf{vec}(\boldsymbol{X}_{\text{true}})\|_\infty}{\|\mathsf{vec}(\boldsymbol{X}_k - \boldsymbol{X}_{\text{true}})\|_2 / \sqrt{MN}}\right)$$

You should see a dramatic rise in the PSNR, from about 25dB to over 50dB.

(g) [5] Make figure showing $\boldsymbol{X}_{\text{true}}$, $\boldsymbol{X}_0$, $\hat{\boldsymbol{X}}$ and the corresponding error images $\boldsymbol{X}_0 - \boldsymbol{X}_{\text{true}}$, $\hat{\boldsymbol{X}} - \boldsymbol{X}_{\text{true}}$.
You should see that the error is reduced dramatically.

Optional problem(s)

3. [0] **Sparsity regularizers**

Challenge. Consider the following two optimizion formulations for transform sparsity:

$$\hat{\boldsymbol{x}}_0 = \arg\min_{\boldsymbol{x}} \Phi_0(\boldsymbol{x}), \quad \Phi_0(\boldsymbol{x}) \triangleq f(\boldsymbol{x}) + \beta\,\|\boldsymbol{T}\boldsymbol{x}\|_1$$

$$\hat{\boldsymbol{x}}_\alpha = \arg\min_{\boldsymbol{x}} \Phi(\boldsymbol{x};\alpha), \quad \Phi(\boldsymbol{x};\alpha) \triangleq f(\boldsymbol{x}) + \beta R_\alpha(\boldsymbol{x}), \quad R_\alpha(\boldsymbol{x}) = \frac{1}{\alpha}\left(\min_{\boldsymbol{z}} \frac{1}{2}\|\boldsymbol{T}\boldsymbol{x} - \boldsymbol{z}\|_2^2 + \alpha\,\|\boldsymbol{z}\|_1\right),$$

where $\beta > 0$ and $\alpha > 0$. Assume $f(\boldsymbol{x})$ is convex. You may also assume that $\hat{\boldsymbol{x}}_0$ and $\hat{\boldsymbol{x}}_\alpha$ are unique minimizers.

Prove, or disprove this conjecture: $\lim_{\alpha\to 0} \hat{\boldsymbol{x}}_\alpha = \hat{\boldsymbol{x}}_0$.