

Homework #5, EECS 598-006, W20. Due **Thu. Feb. 13**, by 4:00PM

This HW looks long, but it takes far more lines of text to describe the autograder problems than to implement the code, and there are a lot of very similar parts so you will be able to recycle a lot of code from one part to another. It is set up this way so that you can use the autograder to check intermediate stages of your work.

1. [3] Cost functions for low-rank plus sparse models

Consider the matrix sensing measurement model $\mathbf{y} = \mathcal{A}(\mathbf{X}) + \varepsilon$, where the latent $M \times N$ matrix \mathbf{X} is thought to be the sum of two matrices, *i.e.*, $\mathbf{X} = \mathbf{L} + \mathbf{S}$, where $\text{rank}(\mathbf{L}) \leq K$ is a low-rank part, and \mathbf{S} is expected to be sparse. The linear operator \mathcal{A} maps $M \times N$ matrices into a vector of length J and $\varepsilon \in \mathbb{F}^J$ denotes additive white Gaussian noise.

Write down a **cost function** and **optimization problem** for forming an estimate $\hat{\mathbf{X}}$ of \mathbf{X} , where the cost function should use the stated signal model properties. Annotate your cost function to explain where your solution captures the different properties. Your cost function must be in a form that does not require an **SVD** operation to solve.

2. [3] Composition of Lipschitz continuous functions

Let $f : \mathbb{F}^M \mapsto \mathbb{F}^K$ and $g : \mathbb{F}^N \mapsto \mathbb{F}^M$ be **Lipschitz continuous** functions with (best) **Lipschitz constants** L_f and L_g respectively.

- (a) [0] Define $h : \mathbb{F}^N \mapsto \mathbb{F}^K$ by $h(x) \triangleq f(g(x))$ and show that h is Lipschitz continuous with (best) Lipschitz constant $L_h \leq L_f L_g$, or devise a simple counter-example where $L_h > L_f L_g$.
- (b) [3] Prove that $L_h = L_f L_g$ when $h = f \circ g$, where \circ denotes **function composition**, or devise a simple counter-example where $L_h < L_f L_g$.

3. [28] Preconditioned steepest descent for smooth convex cost functions

The GD method was easy to implement, but is undesirably slow, so we move towards faster optimization methods. This problem focuses on the **preconditioned steepest descent (PSD)** method. An EECS 551 HW problem required implementing PSD for a quadratic cost function. Here we generalize to any cost function Ψ that is a **convex function** with a **Lipschitz continuous** gradient $\mathbf{g}(\mathbf{x}) = \nabla \Psi(\mathbf{x})$, with known **Lipschitz constant** L_g . This problem is a warm-up for a subsequent PCG problem.

The trickiest part of PSD is the line search step:

$$\alpha_k = \arg \min_{\alpha} h_k(\alpha), \quad h_k(\alpha) \triangleq \Psi(\mathbf{x}_k + \alpha \mathbf{d}_k)$$

where the **search direction** at the k th iteration is denoted

$$\mathbf{d}_k = -P \nabla \Psi(\mathbf{x}_k) = -P \mathbf{g}(\mathbf{x}_k).$$

- (a) [3] Show that $h_k(\alpha)$ is convex.
- (b) [3] Show that $h_k(\alpha)$ has a Lipschitz continuous derivative and find an expression for the Lipschitz constant of that derivative in terms of L_g and \mathbf{d}_k .
- (c) [10] Because $h_k(\alpha)$ is convex with a Lipschitz continuous derivative, we can apply the GD method to perform the line search (the inner minimization over α). You will use your earlier `gd` function for this. Use the scalar $\alpha_0 = 0$ as the initial guess when applying GD for the line search in this and all subsequent problems, unless otherwise specified; this may not be the optimal choice but it is convenient for auto-grading.

Write a JULIA function `psd` that implements the PSD iteration given a function \mathbf{g} that computes the cost function gradient, the Lipschitz constant L_g and an initial guess \mathbf{x}_0 . Your function should take an optional named argument `ninner` that specifies how many inner iterations of GD to use for the line-search step.

Your function should take the usual optional named argument `fun` for evaluating `fun(x, iter)` each iteration.

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

```
"""
    (x,out) = psd(g::Function, Lg::Real, x0::AbstractVector ;
        niter::Int=100, ninner::Int=10, P=I, fun::Function = (x,iter) -> undef)
```

```

Perform preconditioned steepest descent (PSD)
to minimize a convex cost function having a Lg-Lipschitz smooth gradient.

In
* `g` function that computes gradient `g(x)` of a convex cost function
* `Lg` Lipschitz constant of cost function gradient
* `x0` initial guess

Option
* `niter` # number of outer PSD iterations; default `100`
* `ninner` # number of inner iterations of GD for line search; default `10`
* `P` preconditioner; default `I`
* `fun` User-defined function to be evaluated with two arguments `(x,iter)`.
    It is evaluated at `(x0,0)` and then after each iteration.

Out
* `x` final iterate
* `out` `[fun(x0,0), fun(x1,1), ..., fun(x_niter,niter)]`
"""
function psd(g::Function, Lg::Real, x0::AbstractVector ;
    niter::Int=100,
    ninner::Int=10,
    P=I,
    fun::Function = (x,iter) -> undef)

```

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

Hint. Your `gd` function returns two arguments but you probably care only about the first argument, so use something like

```
(alpha, _) = gd( ... )
```

If you followed the template, your previous `gd` function included the following argument:

```
x0::Union{Number,AbstractVector{<:Number}}. We used that Union because in this problem you will
call gd with a scalar initial value for the line search, because  $\alpha$  is scalar.
```

- (d) [3] Write a script that applies both the new `psd` algorithm and the previous `gd` algorithm to solve the *regularized* LS problem $\hat{x} = \arg \min_x \frac{1}{2} \|Ax - y\|_2^2 + \beta \frac{1}{2} \|x\|_2^2$ with $\beta = 5$ for the following data. Initialize with $x_0 = \mathbf{0}$ and use the default $P = I$ preconditioner. (This is a small test to confirm that your code works and to verify that it is faster than GD.)

```
seed!(0); M = 100; N = 50; A = randn(M,N); y = randn(M)
```

Your script should include all the code needed to make the plots in the next two parts.

Your script should call your `psd` and `gd` function exactly once each, using an appropriate `fun`, to get all of the quantities needed for making these plots.

Submit a screenshot of your script to [gradescope](#) to confirm efficiency.

- (e) [3] Plot the \log_{10} cost function $\log_{10}(\Psi(x_k) - \Psi(\hat{x}))$ versus iteration k for both GD and SD on the same axes for $k = 0, \dots, 400$. To avoid errors about log of complex numbers, use `mylog = x -> log10(max(x, 1e-16))` where the `1e-16` comes from `eps(Float64)`.
- (f) [3] Plot the \log_{10} NRMSD to the solution $\log_{10}(\|x_k - \hat{x}\| / \|\hat{x}\|)$ versus iteration k for both GD and SD on the same axes for $k = 0, \dots, 400$.
- (g) [3] You should see that SD converges noticeably faster in your plots. Explain why briefly from a theoretical perspective. Optional: quantify the asymptotic convergence rate here.
- (h) [0] Optional. Are the above plots a fair comparison of GD and SD?

4. [19] PSD for smooth inverse problems

The generic PSD method in the previous problem is applicable to any smooth convex cost function, but it is inefficient for large-scale inverse problems that have the general cost function $\Psi(\mathbf{x}) = \sum_{j=1}^J f_j(\mathbf{B}_j \mathbf{x})$ discussed in the course notes. This problem makes a PSD algorithm that is suitable for this broad class of cost functions, assuming each f_j function is **convex** and has a **Lipschitz continuous** gradient. This problem is a warm-up for a subsequent PCG problem.

The key here is to implement the line-search step efficiently, following the course notes.

- (a) [0] As in the course notes, let $h_k(\alpha) \triangleq \sum_{j=1}^J f_j(\mathbf{u}_j^{(k)} + \alpha \mathbf{v}_j^{(k)})$. Let L_j denote the Lipschitz constant of the gradient of f_j . Recall that you found a Lipschitz constant for the derivative of h_k in a previous HW problem.
- (b) [10] Because $h_k(\alpha)$ is convex with a Lipschitz continuous derivative, we can apply the GD method to perform the line search (the inner minimization over α). Use your earlier `gd` function for this.

Write a JULIA function `psd_inv` that implements the PSD iteration given: an array containing the matrices (or matrix-like objects) $\mathbf{B}_1, \dots, \mathbf{B}_J$, an array of functions for computing the gradients of each f_j , i.e., $\nabla f_1, \dots, \nabla f_J$, an array containing the Lipschitz constants L_1, \dots, L_J , and an initial guess \mathbf{x}_0 .

Note that your function `psd_inv` does not require the Lipschitz constant of $\nabla \Psi$.

Your function should take an optional named argument `ninner` that specifies how many inner iterations of GD to use for the line-search step.

Your function should take the usual optional named argument `fun` for evaluating `fun(x, iter)` each iteration.

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

```
"""
    (x,out) = psd_inv(B::AbstractVector{<:Any},
                    gf::AbstractVector{<:Function}, Lgf::AbstractVector{<:Real},
                    x0::AbstractVector{<:Number} ; niter::Int=100, ninner::Int=10,
                    P=I, fun::Function = (x,iter) -> undef)

Preconditioned steepest descent algorithm
to minimize a general "inverse problem" cost function `sum_{j=1}^J f_j(B_j x)`
where each `f_j` has a `Lgf_j`-Lipschitz smooth gradient.

In
* `B` array of `J` blocks `B_1,...,B_J`
* `gf` array of `J` functions for computing gradients of `f_1,...,f_J`
* `Lgf` array of `J` Lipschitz constants for those gradients
* `x0` initial guess

Option
* `niter` # number of outer PSD iterations; default `100`
* `ninner` # number of inner iterations of GD for line search; default `10`
* `P` preconditioner; default `I`
* `fun` User-defined function to be evaluated with two arguments `(x,iter)`.
It is evaluated at (x0,0) and then after each iteration.

Out
* `x` final iterate
* `out` `[fun(x0,0), fun(x1,1), ..., fun(x_niter,niter)]`
"""
function psd_inv(
    B::AbstractVector{<:Any},
    gf::AbstractVector{<:Function},
    Lgf::AbstractVector{<:Real},
    x0::AbstractVector{<:Number} ;
    niter::Int = 100,
    ninner::Int = 10,
    P = I,
```

```
fun::Function = (x,iter) -> undef)
```

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

- (c) [3] Write a script that applies both the new `psd_inv` algorithm and the previous `gd` and `psd` algorithms to solve the *regularized* LS problem $\hat{x} = \arg \min_x \frac{1}{2} \|Ax - y\|_2^2 + \beta \frac{1}{2} \|x\|_2^2$ with $\beta = 5$ for the following data. Initialize with $x_0 = \mathbf{0}$ and use the default $P = I$ preconditioner. (This is just a small test to confirm that your code works and to verify that it is faster than GD.)

```
seed!(0); M = 4000; N = 1000; A = randn(M,N); y = randn(M)
```

Note that this time the problem size is *larger* to better illustrate the timing differences!

Your script should include all the code needed to make the plots in the next two parts.

Your script should call your functions `psd_inv`, `psd` and `gd` exactly once each, using an appropriate `fun`, to get all of the quantities needed for making these plots.

Submit a screenshot of your code to [gradescope](#) to confirm efficiency.

- (d) [3] Run each algorithm for 2-3 iterations, as a “warm up” to force JULIA to compile. Immediately afterwards (in the same script), run each algorithm for 400 iterations, and plot the \log_{10} cost functions $\log_{10}(\Psi(x_k) - \Psi(\hat{x}))$ versus *elapsed wall time* for all three methods. You should see that the new PSD method converges noticeably faster in time. Use your `fun` to keep track of elapsed time, using the `time()` function. Be sure to label the units of your time axes, and your time axis should start at $t = 0$.
- (e) [3] Also plot the \log_{10} NRMSD to the solution $\log_{10}(\|x_k - \hat{x}\| / \|\hat{x}\|)$ versus elapsed wall time for all three algorithms on the same axes.
- (f) [0] Optional. Now are the above plots a fair comparison?
- (g) [0] Optional. Compare GD and your new SD for the image denoising cost function considered in previous HW.

5. [19] Nonlinear CG for smooth convex cost functions

The **nonlinear conjugate gradient (CG)** method can converge much faster than PSD because it chooses better search directions. This problem implements a general PCG method for a **convex function** Ψ with a **Lipschitz continuous** gradient $g(x) = \nabla \Psi(x)$, with known **Lipschitz constant** L_g . You should start with your general `psd` code (after it passes the autograder) and make a fairly small modification to it.

- (a) [10] Write a JULIA function `ncg` that implements the CG iteration given a function `g` that computes the cost function gradient, the Lipschitz constant L_g and an initial guess x_0 . See template below for all the options. As mentioned in the Fessler book Ch. 11 (and the wikipedia page linked above), there are many methods for choosing the CG β factor that controls the search direction. Your function must implement the (default) Dai-Yuan choice. (Other choices are optional.) Be sure to implement the *preconditioned* version from the reading, not the unpreconditioned one on wikipedia. Your file should be named `ncg.jl` and should contain the following function:

```
"""
    (x,out) = ncg(g::Function, Lg::Real, x0::AbstractVector ;
    niter::Int=100, ninner::Int=10, P=I,
    betahow::Symbol=:dai_yuan, fun::Function = (x,iter) -> undef)

Perform nonlinear (preconditioned) conjugate gradient (PCG)
to minimize a convex cost function having a `Lg`-Lipschitz smooth gradient.

In
* `g` function that computes gradient `g(x)` of a convex cost function
* `Lg` Lipschitz constant of cost function gradient
* `x0` initial guess

Option
* `niter` # number of outer PCG iterations; default `100`
* `ninner` # number of inner iterations of GD for line search; default `10`
* `P` preconditioner; default `I`
* `betahow` "beta" method for the search direction; default `:dai_yuan`
* `fun` User-defined function to be evaluated with two arguments `(x,iter)`.
    It is evaluated at `(x0,0)` and then after each iteration.

Out
* `x` final iterate
* `out` [fun(x0,0), fun(x1,1), ..., fun(x_niter,niter)]
"""
function ncg(g::Function, Lg::Real, x0::AbstractVector ;
    niter::Int = 100,
    ninner::Int = 10,
    P = I,
    betahow::Symbol = :dai_yuan,
    fun::Function = (x,iter) -> undef)
```

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

- (b) [3] Write a JULIA script that applies your JULIA functions `ncg` and `psd` to solve the regularized LS problem $\hat{x} = \arg \min_x \frac{1}{2} \|Ax - y\|_2^2 + \beta \frac{1}{2} \|x\|^2$ for the following data with $\beta = 5$. Initialize with $x_0 = \mathbf{0}$ and use the default $P = I$.
- ```
seed!(0); M = 100; N = 50; A = randn(M,N); y = randn(M)
```
- You should call your functions exactly once, using an appropriate `fun`, to get all of the quantities needed for making the following plots.
- Submit a screenshot of your code to [gradescope](#) to confirm efficiency.
- (c) [3] Run both methods for 200 iterations and (as usual) plot  $\log_{10}(\Psi(x_k) - \Psi(\hat{x}))$  versus iteration  $k$  for both methods on one axes. Use `mylog` again as above.
- (d) [3] Also make a plot of  $\log_{10}(\|x_k - \hat{x}\| / \|\hat{x}\|)$  versus iteration  $k$  for both methods.
- (e) [0] Compare quantitatively the asymptotic convergence rates.

The next HW will develop a version of PCG for inverse problems, analogous to `psd_inv`, and compare its wall time to the other algorithms for both a test case and for a larger image denoising application. It would be logical to have that problem here as the “grand finale” of the PSD/PCG sequence, but this HW seemed long enough already.

---

### In-class task problem(s)

---

The following problem(s) were started in class. They are repeated here for completeness because they are due with this HW.

#### 6. [33] 2D finite differences efficiently

A previous HW did 1D signal denoising using a first-order finite-difference matrix created using one of these two commands:

```
D = spdiags(0 => -ones(N-1), 1 => ones(N-1))
D = spdiags(0 => -ones(N-1), 1 => ones(N-1)) [1:end-1, :]
```

In this work we will use the latter form because the former has an unnecessary (though harmless) extra row of zeros at the end.

To work with 2D images of size  $M \times N$  (instead of 1D signals) we need often use regularization based on the following finite difference matrix:

$$C = \begin{bmatrix} I_N \otimes D_M \\ D_N \otimes I_M \end{bmatrix}, \quad (1)$$

where  $D_N$  denotes the  $(N-1) \times N$  matrix created by the 2nd JULIA command above.

Here you will first create the matrix  $C$  in JULIA using a sparse array, and then you will make a more efficient version using the `LinearMapsAA` package.

(a) [10] Write a JULIA function that creates the sparse matrix in (1) given the image dimensions  $M \times N$ .

Hint. Use the `spdiags` function in the `SparseArrays` package, the `kron` function, and `I(n)`.

Your file should be named `diff2d_sp.jl` and should contain the following function:

```
"""
 C = diff2d_sp(M::Int, N::Int)

Create a sparse matrix for computing first-order finite differences
along both dimensions. Mathematically:
`C = [(I_N \otimes D_M); (D_N \otimes I_M)]`
where `D_N` denotes the `(N-1) x N` 1D finite difference matrix
and `\otimes` denotes the Kronecker product.

In
* `M,N` 2D image size

Out
* `C` sparse matrix of size `(N*(M-1) + (N-1)*M) x (M*N)`
"""
function diff2d_sp(M::Int, N::Int)
```

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

(b) [10] The sparse matrix approach is expedient for medium-sized problems, but is inefficient for large problems. So now we move towards a more efficient approach that may *not* use any `SparseArrays` functions like `spdiags`.

First we need a JULIA function that computes  $d = C \text{vec}(x)$  directly, without any explicit matrix operations.

Write a JULIA function `diff2d_forw` that takes as input a  $M \times N$  array `x` and returns the  $N \cdot (M-1) + (N-1) \cdot M$  length vector  $d = C \text{vec}(x)$ . Hint. The `diff` function is useful. It works for 2D arrays with an optional 2nd argument!

Your file should be named `diff2d_forw.jl` and should contain the following function:

```
"""
 d = diff2d_forw(x::AbstractMatrix)

Compute 2D finite differences along both dimensions.
Performs the same operations as
`d = [(I_N \otimes D_M); (D_N \otimes I_M)] x[:]'
where `D_N` denotes the `(N-1) x N` 1D finite difference matrix
and `\otimes` denotes the Kronecker product,
but does it efficiently without using any `SparseArrays` functions.
"""
```

```

In
* `x` `M x N` array (typically a 2D image).
It cannot be a `Vector`! (But it can be a `Mx1` or `1xN` 2D array.)

Out
* `d` vector of length `N*(M-1) + (N-1)*M`
"""
function diff2d_forw(x::AbstractMatrix)

```

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

- (c) [10] Next we need a JULIA function that computes the effect of multiplying by the transpose  $z = C'd$ , but without any matrix operations. This is also called the **adjoint** operation.

Write a JULIA function `diff2d_adj` that takes as input a vector `d` of length  $N \cdot (M - 1) + (N - 1) \cdot M$ , along with  $M$  and  $N$ , and returns a  $M \times N$  array `z` corresponding to (a reshaped version of)  $z = C'd$ .

Keep in mind that adjoint is not the same as inverse!

Hint. My solution uses `transpose` a couple times because `array[1, :]` produces a column vector, not a row vector like in MATLAB. Do not use Hermitian transpose `'` or your code will fail for complex inputs.

Hint. Using properties of the transpose of a block matrix and of the Kronecker product, the transpose of  $C$  in (1) is

$$C' = [I_N \otimes D'_M \quad D'_N \otimes I_M].$$

The `diff1` notebook discussed in class showed how to implement multiplication by  $D'_N$  so you can adapt that code. My solution uses `reshape` and `[:]` and indeed those operations arise frequently in image processing applications.

Your file should be named `diff2d_adj.jl` and should contain the following function:

```

"""
 z = diff2d_adj(d::AbstractVector{<:Number}, M::Int, N::Int ; out2d=false)

Compute adjoint of 2D finite differences along both dimensions.
Performs the same operations as
`z = [(I_N \otimes D_M); (D_N \otimes I_M)]' * d`
where `D_N` denotes the `(N-1) x N` 1D finite difference matrix
and `\otimes` denotes the Kronecker product, but does it
efficiently without using any `SparseArrays` functions.

In
* `d` vector of length `N*(M-1) + (N-1)*M`
* `M,N` desired output size

Option
* `out2d::Bool` if `true` return `M x N` array, else `M*N` vector; default `false`

Out
* `z` `M*N` vector or `M x N` array (typically a 2D image)
"""
function diff2d_adj(
 d::AbstractVector{<:Number}, M::Int, N::Int ; out2d::Bool=false)

```

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

- (d) [0] Create a `LinearMapAA` object that uses your `diff2d_forw` and `diff2d_adj` functions using the command

`Clm = LinearMapAA( ... )` with appropriate arguments, for a test image of size  $(M, N) = 4, 5$ .

Verify yourself that your object is correct by first creating a sparse matrix `Csp` in JULIA corresponding to (1) using

`Csp = diff2d_sp(M,N)`, and then typing `Matrix(Csp) == Matrix(Clm)`

If this check does not return `true` then look at `Matrix(Csp)` and `Matrix(Clm)` for small values of  $M$  and  $N$  to debug your code.

- (e) [0] Now test your adjoint function by typing these commands:

```
Matrix(Csp') == Matrix(Clm')
```

```
Matrix(Clm') == Matrix(Clm)'
```

When these checks pass, then congratulations! You now have working code for computing 2D finite differences of large images efficiently.

(f) [3] Test your `LinearMapAA` object visually as follows:

```
using Plots
using LinearMapsAA
using MIRT: jim, ellipse_im
include("diff2d_ans.jl") # use your file's name

M,N = 60,64
xtest = ellipse_im(M,N)

forw = x -> diff2d_forw(reshape(x,M,N))
adj = d -> diff2d_adj(d,M,N)
Clm = LinearMapAA(forw, adj, (N*(M-1) + (N-1)*M, M*N))

CtCx = reshape(Clm'*(Clm*xtest[:]), M,N)
plot(jim(xtest, "x"), jim(CtCx, "C'*C*x"))
savefig("diff2d-see.pdf")
```

Think about whether the resulting image makes sense and upload it to [gradescope](#).

- At this point we do not have any way to autograde your `LinearMapAA` object, but a future HW problem will use it for an image processing application. You may even use it for HW4 if you want.
- If you finish early, then please help others at your table or nearby.
- Optional challenge. The methods here only provide horizontal and vertical finite differences. It can be helpful to also use diagonal finite differences. Think about how you would implement that.