Homework #6, EECS 598-006, W20. Due **Thu. Feb. 21**, by 4:00PM

---

1. [22] **Nonlinear CG for smooth inverse problems**

The generic nonlinear CG (NCG) method in a previous HW problem is applicable to any smooth convex cost function, but it is inefficient for large-scale inverse problems that have the general cost function $\Psi(\boldsymbol{x}) = \sum_{j=1}^{J} f_j(\boldsymbol{B}_j \boldsymbol{x})$ discussed in the course notes. In this problem you will make a NCG 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 not a warm-up: this is finally a very useful algorithm for large-scale signal and image processing problems.

The key here is to implement the line-search step efficiently, following the steps you used in your `psd_inv` code, and to use the search direction from your `ncg_inv` code.

(a) [10] Write a JULIA function `ncg_inv` that implements the NCG iteration given the same inputs used in your `psd_inv` code. Your function should implement the same choice of the CG β factor as your generic `ncg` algorithm.

Importantly, your function `ncg_inv` does not require the Lipschitz constant of $\nabla \Psi$.

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

```
"""
    (x,out) = ncg_inv(B, gf, Lgf, x0 ; [options])

Nonlinear preconditioned conjugate gradient 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 iterations; default `50`
* `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_inv(
        B::AbstractVector{<:Any},
        gf::AbstractVector{<:Function},
        Lgf::AbstractVector{<:Real},
        x0::AbstractVector{<:Number} ;
        niter::Int = 50,
        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 script that applies both the new `ncg_inv` algorithm and the previous `ncg` and `psd_inv` algorithms to

solve the *regularized* LS problem $\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}} \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \beta \frac{1}{2}\|\boldsymbol{x}\|_2^2$ with $\beta = 5$ for the following data. Initialize with $\boldsymbol{x}_0 = \boldsymbol{0}$ and use the default $\boldsymbol{P} = \boldsymbol{I}$ preconditioner. (This is just a medium-sized test to confirm that your code works and to verify that it is faster than the generic nonlinear CG.)

```
seed!(0); M = 4000; N = 1000; 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 functions `psd_inv`, `ncg` and `ncg_inv` 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.

(c) [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 60 iterations, and plot the log10 cost functions $\log10(\Psi(\boldsymbol{x}_k) - \Psi(\hat{\boldsymbol{x}}))$ versus *elapsed wall time* for all three methods. You should see that the new NCG method converges noticeably faster in time.

(d) [3] Also plot the log10 NRMSD to the solution $\log10(\|\boldsymbol{x}_k - \hat{\boldsymbol{x}}\| / \|\hat{\boldsymbol{x}}\|)$ versus elapsed wall time for all three algorithms on the same axes.
A subsequent problem will use this tool for image denoising.
You can probably find generic nonlinear CG packages online, but it is unlikely that you will find something for solving general inverse problems like you will have developed here. (Let me know if you do!) Even the Michigan Image Reconstruction Toolbox (**MIRT**) (for MATLAB) lacks this generality. JULIA makes it easier to write a method that is both efficient and quite general.

(e) [3] Suppose each function f_j is a simple quadratic of the form $f_j(\boldsymbol{u}) = \gamma_j \frac{1}{2}\|\boldsymbol{u} - \boldsymbol{z}_j\|_2^2$ for some positive constants γ_j and vectors $\boldsymbol{z}_j \in \mathbb{R}^{N_j}$. How many inner iterations should one use for the line search? Explain your answer with both words *and* equation(s).

---

2. [19] **OGM with a line search for smooth inverse problems**

The generic OGM is applicable to any smooth convex cost function, but it is inefficient for large-scale inverse problems that have the general cost function $\Psi(\boldsymbol{x}) = \sum_{j=1}^{J} f_j(\boldsymbol{B}_j\boldsymbol{x})$. This problem makes a line-search version of OGM that is suitable for this broad class of cost functions, assuming each f_j function is **convex** and has a **Lipschitz continuous** gradient.

This course is designed to prepare you to be able to read modern optimization algorithms in the literature and understand their properties and implement them. You have implemented several algorithms from course notes, and now it is time to pursue something truly recent. This very recent paper http://doi.org/10.1007/s10107-019-01410-2 describes the OGM line-search approach on p. 22, and the definition of θ_i is a couple pages earlier. (You need not study the whole paper.)

Think about efficiency when implementing this algorithm! Use recursive updates similar to `psd_inv` and `ncg_inv`. Also, one term in the algorithm is $\sum_{j=0}^{i-1} \theta_j \nabla\Psi(\boldsymbol{x}_j)$. Do not write a loop to recompute this sum every iteration. Compute this sum incrementally by using `+=` to add the latest gradient to a running sum each iteration.

My solution uses 5 loops over J each iteration, one of which updates $\boldsymbol{B}_j\boldsymbol{x}_i$ and one of which updates $\boldsymbol{B}_j\boldsymbol{y}_i$. If you find an efficient way to do it with 4 or fewer loops please let me know! As usual, the key here is to implement the line-search step efficiently, following the steps used in earlier problems. You should be able to reuse a lot of your previous code.

(a) [10] Write a JULIA function `ogm_inv` that implements the line-search OGM from the above paper, given the same inputs used in your `psd_inv` code. Note that your function `ogm_inv` does not require the Lipschitz constant of $\nabla\Psi$.
Your file should be named `ogm_inv.jl` and should contain the following function:

```
"""
    (x,out) = ogm_inv(B, gf, Lgf, x0 ; niter=?, ninner=?, fun=?)

OGM with line search; Drori&Taylor http://doi.org/10.1007/s10107-019-01410-2
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.
Uses 1D GD for the line search.

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 iterations; default 50
* `ninner`    # number of inner iterations of GD for line search; default 10
* `fun`        User-defined function to be evaluated with 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 ogm_inv(
        B::AbstractVector{<:Any},
        gf::AbstractVector{<:Function},
        Lgf::AbstractVector{<:Real},
        x0::AbstractVector{<:Number} ;
        niter::Int=50,
        ninner::Int=10,
        fun::Function = (x,iter) -> undef)
```

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

(b) [3] Write a script that applies both the new `ogm_inv` algorithm and the previous `ncg_inv` algorithms to solve the regularized LS problem $\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}} \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \beta\frac{1}{2}\|\boldsymbol{x}\|_2^2$ with $\beta = 5$ for the following data. Initialize with $\boldsymbol{x}_0 = \boldsymbol{0}$ and use the default $\boldsymbol{P} = \boldsymbol{I}$ preconditioner for CG. (This is just a medium-sized test to confirm that your code works and to compare it to nonlinear CG.)

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

Your script should include all the code needed to make the plots in the next two parts. As usual, your script should call your functions exactly once each to get all of the quantities needed for making these plots.
Submit a screenshot of your code to gradescope to confirm efficiency.

(c) [3] Run each algorithm for 2 iterations, as a "warm up" to force JULIA to compile. Immediately afterwards (in the same script), run each algorithm for 30 iterations, and plot the log10 cost functions $\log_{10}(\Psi(\boldsymbol{x}_k) - \Psi(\hat{\boldsymbol{x}}))$ versus elapsed wall time for both methods. You should see that OGM starts out like CG but eventually lags, which is unsurprising because CG is especially useful for quadratic problems.

(d) [3] Also plot the log10 NRMSD to the solution $\log_{10}(\|\boldsymbol{x}_k - \hat{\boldsymbol{x}}\| / \|\hat{\boldsymbol{x}}\|)$ versus elapsed wall time for both algorithms.

(e) [0] Optional. Also plot the cost and NRMSD versus iteration to eliminate the effects of possibly inefficient implementations. (For my implementation, OGM is only a tiny bit slower than CG per iteration.)

3. [22] **Edge-preserving image denoising using fast algorithms**

This problem investigates the edge-preserving **image denoising** application, where we want to recover an image $\boldsymbol{x}$ from noisy measurements $\boldsymbol{y}$ under the model $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{\varepsilon}$ using the optimization problem

$$\hat{\boldsymbol{x}} = \underset{\boldsymbol{x} \in \mathbb{C}^N}{\arg\min}\, \Psi(\boldsymbol{x}), \quad \Psi(\boldsymbol{x}) = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{x}\|_2^2 + \beta R(\boldsymbol{x}), \quad R(\boldsymbol{x}) = \sum_k \psi([\boldsymbol{C}\boldsymbol{x}]_k, \delta),$$

where ψ denotes the **Fair potential**, and $\boldsymbol{C}$ denotes the 2D first-order finite-differencing matrix. For simplicity, we focus on the case of real-valued images here. You will compare two optimization algorithms developed in a previous problems: the efficient CG algorithm `ncg_inv` and the OGM line-search algorithm `ogm_inv`.

(a) [10] Write a JULIA function `dn2cg` that uses your `ncg_inv` or your `ogm_inv` code to minimize the above cost function. Your function must work for large-scale problems, so it cannot use expensive and memory hungry operations like `svd` `svdvals` `eigen` `eigvals` `opnorm` etc. And for further speed, it must use `LinearMapAA` for $\boldsymbol{C}$, instead of `spdiagm` `sparse` `kron`. The prototype below provides the specifications.

Your file should be named `dn2cg.jl` and should contain the following function:

```
"""
    (x,cost,out) = dn2cg(y ; x0=?, [options])

Performs 2D edge-preserving image denoising using either the NCG
algorithm or the OGM line-search algorithm, to "solve" the minimization problem
`argmin_x 1/2 |y - x|^2 + reg * sum_k pot([C x]_k,del)`
where `pot()` is the Fair potential with parameter `del`
and `C` denotes the 2D first-order finite differencing matrix.

Uses these functions from previous problems:
     `ncg_inv ogm_inv diff2d_forw diff2d_adj`

May not use any `SparseArrays` functions, SVD, etc.

In
* `y`        2D noisy real grayscale image of size `[M,N]`

Option
* `how`        `:cg` (default) to use CG or `:ogm` to use OGM
* `x0`        2D initial guess of size `[M,N]`; default `y`
* `reg`        regularization parameter; default 1
* `del`        potential function parameter; default 2
* `P`        preconditioner for CG; default `I`
* `niter`    # number of iterations; default 100
* `ninner`    # number inner GD iterations for line search; default 10
* `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`    2D final iterate image of size `[M,N]`
* `cost`    [niter+1]` cost function each iteration
* `out`  [fun(x0,0), fun(x1,1), ..., fun(x_niter,niter)]
"""
function dn2cg(y::AbstractMatrix ;
        how::Symbol = :cg,
        x0::AbstractMatrix = y,
        reg::Real = 1,
        del::Real = 2,
        P=I,
        niter::Int = 100,
```

```
        ninner::Int = 10,
        fun::Function = (x,iter) -> undef)
```

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

(b) [3] Write a JULIA script that applies your 2D image denoising method `dn2cg` to the 2D noisy image generated by the following code, using 20 iterations of CG.

```
using Random: seed!
tmp = [
    zeros(1,20);
    0 1 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1 1 1 0;
    0 1 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 1 0 0;
    0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0;
    0 0 1 1 1 1 0 0 0 0 1 1 0 0 0 0 0 1 1 0;
    zeros(1,20)
]';
xtrue = kron(10 .+ 80*tmp, ones(13,13)) # note bigger size!
seed!(0)
y = xtrue + 20 * randn(size(xtrue))
```

Submit a screenshot of your script code for the next part to gradescope.

(c) [3] Make a figure showing the original image $\boldsymbol{x}_{\text{true}}$, the noisy data $\boldsymbol{y}$, and the denoised images $\hat{\boldsymbol{x}}$ for both $(\beta, \delta) = (5, 100)$ and $(\beta, \delta) = (5, 3)$. The case $\delta = 100$ is essentially the same as quadratic regularization, which causes edge blurring, whereas the small value of δ provides edge-preserving regularization. Label the image $\boldsymbol{y}$ and each $\hat{\boldsymbol{x}}$ with their NRMSE values $\|\hat{\boldsymbol{x}} - \boldsymbol{x}_{\text{true}}\| \,/\, \|\boldsymbol{x}_{\text{true}}\|$.

(d) [0] Optional: adopt the generic code for FGM and/or OGM (with gradient-based restart) online here:
https://gitlab.eecs.umich.edu/michigan-fast-optimization/ogm-adaptive-restart
for this denoising application.

(e) [3] Write a script that applies 20 iterations of CG and OGM line-search with $(\beta, \delta) = (5, 3)$ to the same noisy image. Submit a screenshot of your script code for the next part to gradescope.

(f) [3] Plot $\log_{10}(\Psi(\boldsymbol{x}_k))$ for both CG and OGM line-search vs k and vs elapsed time. Be sure to "warm up" your algorithms so that time starts at zero. Optionally compared to the generic FGM/OGM.

---

4. [9] **OGM for $\boldsymbol{S}$-Lipschitz continuous gradients**

The **optimized gradient method** (OGM) presented in the course notes and in the original paper http://doi.org/10.1007/s10107-015-0949-3 is for a cost function $f : \mathbb{R}^N \mapsto \mathbb{R}$ whose gradient is smooth with Lipschitz constant L.
Suppose cost function $\Psi : \mathbb{R}^N \mapsto \mathbb{R}$ is known to have a $\boldsymbol{S}$-Lipschitz continuous gradient, for some invertible matrix $\boldsymbol{S}$.

(a) [3] Apply the change of variables method presented in the course notes to derive (easily!) a generalized version of OGM that uses the matrix $\boldsymbol{S}$ instead of the constant L.
Hint. Your generalized OGM should revert to the original OGM when $\boldsymbol{S} = \sqrt{L}\boldsymbol{I}$.

(b) [3] For your generalized OGM, give a useful upper bound on the cost function gap $\Psi(\boldsymbol{x}_n) - \Psi(\boldsymbol{x}_\star)$ in terms of $\boldsymbol{S}$ and other relevant quantities. Explain briefly why your bound is correct.

(c) [3] Is your bound for your generalized OGM tight? Explain why or why not.

5. [3] **Best Lipschitz constant for edge-preserving regularizer**

As discussed in Ch. 3, a typical edge-preserving regularizer has the form $R(\boldsymbol{x}) = \mathbf{1}' \, \psi \, . (\boldsymbol{C}\boldsymbol{x})$, where $\boldsymbol{C}$ is a $K \times N$ matrix and $\boldsymbol{x} \in \mathbb{R}^N$. Assume that ψ satisfies Huber's conditions with $\omega_\psi(0) = \lim_{r \to 0} \dot{\psi}(r) \, / r$ and assume that $L_{\dot{\psi}} = \omega_\psi(0)$. Show that the best Lipschitz constant for ∇R is

$$L_{\nabla R} = L_{\dot{\psi}} \| \boldsymbol{C}' \boldsymbol{C} \|_2 .$$

Hint. As one possible approach, take $\boldsymbol{z} = \mathbf{0}$ and $\boldsymbol{x} = \epsilon \boldsymbol{v}_1$ where $\boldsymbol{v}_1$ denotes the first right singular vector of $\boldsymbol{C}$.

Optional challenge: prove or disprove when ψ does not necessarily satisfy Huber's conditions.

6. [10] Make a clearly stated problem that is suitable for an exam in this course, based on the material in Ch. 1-3 (HW 1-6), and provide your own solution to the problem. Your problem could include something about JULIA, but in that case it should be about concepts, not mere syntax, and should not require submitting code to an autograder.
If you make a good problem (not too trivial, not too hard) then it might be used on an actual exam. To earn full credit:
- Your problem and solution must fit on a single page with a horizontal line that clearly separates the question (above) from the answer (below).
- Your solution to the problem must be correct.
- Your problem should not be excessively trivial (nor beyond the scope of the course).
  Ideally it should use ideas from this course that go beyond what was covered in EECS 551, although some overlap with the last two chapters of EECS 551 (optimization and matrix completion) is fine.
- Your problem must not be identical to homework or clicker problems from 551 or this course.
- Upload your problem/solution to gradescope as usual for grading.
  Also submit your problem/solution in pdf format to Canvas for distribution by the deadline for this HW.
  We can export from Canvas easily but not from gradescope.
- Do not submit any other parts of your HW assignment to Canvas; only this 1 page pdf file.
- Do not put your name on the problem/solution that you upload to Canvas because we plan to export all problems/solutions to distribute as practice problems.

Optional problem(s)

7. [0] Consider the function $f : \mathbb{C}^N \mapsto \mathbb{R}$ defined by $f(\boldsymbol{x}) = \text{real}\{\boldsymbol{v}'\boldsymbol{x}\}$ for some vector $\boldsymbol{v} \in \mathbb{C}^N$.
Show that $\boldsymbol{d} = -\boldsymbol{v}'$ is a **descent direction** for f, so that it is natural to define $\nabla f(\boldsymbol{x}) = \boldsymbol{v}$.