Homework #8, EECS 598-006, W20. Due **Thu. Mar. 20**, by 4:00PM

1. [19] **Nonlinear CG with MM line search**

Consider the following general "block" cost function

$$\Psi(\boldsymbol{x}) = \sum_{j=1}^{J} f_j(\boldsymbol{B}_j \boldsymbol{x}),\ \boldsymbol{x} \in \mathbb{F}^N,\ \boldsymbol{B}_j \in \mathbb{F}^{M_j \times N},\ f_j : \mathbb{F}^{M_j} \mapsto \mathbb{R}.$$

The earlier nonlinear CG code (`ncg_inv`) assumed that each f_j has a **Lipschitz continuous** gradient, and using that properties we applied GD to perform the **line search** step $\alpha_k = \arg\min_\alpha \Psi(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$. (There we also unnecessarily assumed that each f_j is **convex**.) In this problem you will develop a new version of CG using a line search method that is often *faster*. We will *not* assume each f_j is convex. Instead of assuming that ∇f_j is Lipschitz continuous, we will assume that each $f_j(\boldsymbol{t})$ is **differentiable** and that each has a **quadratic majorizer** of the form

$$q_j(\boldsymbol{t}; \boldsymbol{s}) = f_j(\boldsymbol{s}) + \operatorname{real}\{\langle \nabla f_j(\boldsymbol{s}),\, \boldsymbol{t} - \boldsymbol{s} \rangle\} + \frac{1}{2} (\boldsymbol{t} - \boldsymbol{s})' \operatorname{Diag}\{\boldsymbol{c}_j(\boldsymbol{s})\} (\boldsymbol{t} - \boldsymbol{s}) \geq f_j(\boldsymbol{t}), \tag{1}$$

where $\boldsymbol{c}_j : \mathbb{F}^{M_j} \mapsto \mathbb{R}^{M_j}$ is a vector of nonnegative **curvatures** that, in general, depends on the shape of $f_j(\cdot)$. This assumption allows us to use a **majorize-minimize** (**MM**) approach for the line search, instead of GD, often leading to faster convergence.

(a) [3] For a function f_j that satisfies the assumptions used in the `ncg_inv` code, what is $\boldsymbol{c}_j(\boldsymbol{s})$ in (1) ?

(b) [10] Write a JULIA function that implements the nonlinear CG iteration with MM-based line search, given *almost* the same inputs used in your `ncg_inv` code. The only change is that instead of specifying the Lipschitz constant, now the user supplies functions that compute the curvatures $\boldsymbol{c}_j(\cdot)$.

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

```
"""
    (x,out) = ncg_inv_mm(B, gradf, curvf, x0 ; ...)

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(t)` has a quadratic majorizer of the form
`q_j(t;s) = f_j(t) + \\nabla f_j(s) (t - s) + 1/2 \\|t - s\\|^2_{C(s)}`
where C(s) is diagonal matrix of curvatures, with MM line search.

In
* `B`        array of J blocks `B_1,...,B_J`
* `gradf`    array of J functions for computing gradients of `f_1,...,f_J`
* `curvf`    array of J functions `z -> curv(z)` that return a scalar or
a vector of curvature values for each element of `z`
* `x0`    initial guess; need `length(x) == size(B[j],2)` for `j=1...J`

Option
* `niter` # number of outer iterations; default 50
* `ninner` # number of inner iterations of MM line search; default 5
* `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.

Output
* `x` final iterate
* `out::Array{Any}` `[fun(x0,0), fun(x1,1), ..., fun(x_niter,niter)]`
"""
function ncg_inv_mm(
        B::AbstractVector{<:Any},
        gradf::AbstractVector{<:Function},
        curvf::AbstractVector{<:Function},
        x0::AbstractVector{<:Number};
        niter::Int=50,
        ninner::Int=5,
        P=I,
        betahow::Symbol=:dai_yuan,
        fun::Function = (x,iter) -> undef)
```

Submit your solution to `mailto:eecs556@autograder.eecs.umich.edu`.
Hint. Previously you used GD by calling `gd` to do the inner line search. That worked fine before because we had a fixed step size for the inner line search. Here the inter line search step size varies each inner iteration so you must incorporate ideas from the Ch. 4 notes about quadratic majorizers.

(c) [3] Compare your `ncg_inv` and `ncg_inv_mm` for the regularized LS test problem considered in HW6#1b, as usual. Submit a plot of either $\Psi(\boldsymbol{x}_k)$ or NRMSD versus iteration k (or both if you want) of your comparison to gradescope.

(d) [3] Discuss any similarities or differences that you observe.

(e) [0] Optional. If you are having trouble passing the autograder, try testing your code on a *weighted* LS cost function $\frac{1}{2}\|\boldsymbol{A}\boldsymbol{x}-\boldsymbol{y}\|_{\boldsymbol{W}}^2$ where $\boldsymbol{A}$ has full column rank and $\boldsymbol{W} = \mathsf{Diag}\{\boldsymbol{w}\}$ is a diagonal matrix with positive diagonal elements.

(f) [0] Optional challenge: prove (or disprove with a counter-example) whether (1) implies that ∇f_j is Lipschitz continuous.

2. [25] **Robust regression**

This problem applies the new CG-MM method from the previous problem to the **robust regression** problem

$$\hat{\boldsymbol{x}} = \mathop{\arg\min}_{\boldsymbol{x}\in\mathbb{R}^N} \Psi(\boldsymbol{x}), \qquad \Psi(\boldsymbol{x}) = \mathbf{1}'_M\, \psi.(\boldsymbol{A}\boldsymbol{x}-\boldsymbol{y},\delta) = \sum\nolimits_{i=1}^{M} \psi([\boldsymbol{A}\boldsymbol{x}-\boldsymbol{y}]_i,\delta), \tag{2}$$

where ψ is the Fair potential function with parameter δ, given $\boldsymbol{A}\in\mathbb{R}^{M\times N}$ and $\boldsymbol{y}\in\mathbb{R}^M$.

If we were to let $\delta\to\infty$ in the Fair potential for the above cost function, then $\psi(t,\delta)\to t^2/2$ and we would have

$$\Psi(\boldsymbol{x}) \to \sum\nolimits_{i=1}^{M} \frac{1}{2}([\boldsymbol{A}\boldsymbol{x}-\boldsymbol{y}]_i)^2 = \frac{1}{2}\left\|\boldsymbol{A}\boldsymbol{x}-\boldsymbol{y}\right\|_2^2.$$

In other words, ordinary least-squares regression is a special case of the general cost function (2). However, ordinary LS is not robust to data outliers. Using a Fair potential provides robustness, as you will see in the results below.

(a) [10] Write a script that compares the new `ncg_inv_mm` to the old `ncg_inv` for the robust polynomial fitting problem using the following data that has an outlier.

```
using Plots; default(markerstrokecolor=:auto)
using Random: seed!

s = (t) -> atan(4*(t-0.5)); # nonlinear function

M = 15
seed!(1); tm = sort(rand(M)) # M random sample locations
y = s.(tm) + 0.1 * randn(M); # noisy samples
y[12] = -0.5 # corrupt data with an outlier

deg = 3 # polynomial degree
Afun = (tt) -> [t.^i for t in tt, i in 0:deg] # matrix of monomials
A = Afun(tm) # M × 4 matrix
xz = A \ y # ordinary LS solution

scatter(tm, y, color=:blue,
    label="y", xlabel="t", ylabel="y", ylim=[-1.3, 1.3])
t0 = LinRange(0, 1, 101) # fine sampling for showing curve
plot!(t0, s.(t0), color=:blue, label="s(t)", legend=:topleft)
plot!(t0, Afun(t0)*xz, line=:red, label="ordinary LS cubic fit")
```

Your script should include all the code needed to make the plots in the next two parts. Use $\delta = 0.01$ and initialize $\boldsymbol{x}_0$ with the original LS solution. Your script should call your functions `ncg_inv` and `ncg_inv_mm` exactly once each, using an appropriate `fun`, to get all of the quantities needed.
Submit a screenshot of your code to gradescope to confirm efficiency.

(b) [10] Plot the cost function $\Psi(\boldsymbol{x}_k)$ versus iteration k for both methods, for 90 iterations.
For your plot, use `ninner=9` for `ncg_inv` and `ninner=5` for `ncg_inv_mm`.
(You should experiment with other values yourself, especially for the fair case when both use 5 inner iterations.)

(c) [5] Plot the (corrupted) raw data, the true signal, the ordinary LS fit and the robust LS fit from the 90th iteration of your new CG method. Hint. The robust method should agree with the true signal much better.

3. [22] **Optimal quadratic majorizer for Huber hinge loss**

For training a **binary classifier**, consider the Huber hinge loss function defined as:

$$h(t;\delta) = \begin{cases} 1 - t - \delta/2, & t \leq 1 - \delta \\ \frac{1}{2\delta}(t-1)^2, & 1 - \delta \leq t \leq 1 \\ 0, & 1 \leq t. \end{cases}$$

In class you derived the curvature of the optimal quadratic majorizer for this function for both $1 \leq s$ and $1 - \delta \leq s \leq 1$.

(a) [3] For $s \leq 1 - \delta$, determine the coefficients c_0, c_1, c_2 of the optimal quadratic majorizer

$$q(t;s) = c_0(s) + c_1(s)\,(t - s) + \frac{c_2(s)}{2}(t - s)^2.$$

Your majorizer must satisfy $q(t;s) \geq h(t;\delta),\ \forall t, s \in \mathbb{R}$ and $q(s;s) = h(s;\delta),\ \forall s \in \mathbb{R}$.
Here, “optimal” means c_2 is as small as possible, because smaller curvature value typically means larger step sizes.

(b) [10] Write a JULIA function that returns the coefficient vector $[c_0, c_1, c_2]$ of the *optimal* quadratic majorizer $q(t;s)$ for the Huber hinge, for *any* given value of s and δ.
Hint. Because the Huber hinge is defined piece-wise with braces, your code will likely have `if`-`else` statements.
Your file should be named `huberhinge.jl` and should contain the following function:

```
"""
    coef = huberhinge(s ; delta=?)

Coefficients of optimal quadratic majorizer ``q(t;s)``
for Huber hinge function:
``
h(t; \\delta) = \\leftbrace{
1 - t - \\delta/2, & t \\leq 1 - \\delta \\\\
\\frac{1}{2\\delta} (t - 1)^2, & 1 - \\delta \\leq t \\leq 1 \\
0, & 1 \\leq t }
``


in
* `s` expansion point for which ``q(s;s) = h(s;\\delta)``

option
* `delta 0 < delta,` corner rounding value; default 0.1

out
* `coef` `[c0,c1,c2]` for ``q(t;s) = c0 + c1 (t - s) + c2/2 (t-s)^2``
"""
function huberhinge(s::Real ; delta::Real=0.1)
```

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

(c) [3] Write a script that uses your `huberhinge` function to make a plot, like the one shown in the course notes, of the Huber hinge function and your optimal quadratic majorizer for the case $\delta = 0.4$ and $s = -2$. Label all important points on the horizontal axis. For comparison, include the quadratic majorizer based on the Lipschitz constant of the derivative of h on your plot. Choose axis limits appropriately for clear visualization.
Submit a screenshot of your script to gradescope.

(d) [3] Submit a screenshot of your plot to gradescope.

(e) [3] Revise your script (or make a new one) to make an analogous plot for the case $\delta = 0.4$ and $s = 0.8$.
Submit a screenshot of your plot to gradescope.

The next HW problem uses this tool to train a binary classifier.

4. [20] **Learning a binary classifier for handwritten digits**

This problem explores the machine learning problem of designing a classifier for handwritten digit images. For simplicity we focus on a two-class problem: digit 5 and digit 8. (Those two digits can be somewhat hard to distinguish.) An EECS 551 HW problem used the **logistic loss** function and a quadratic majorizer. Here we use the **Huber hinge loss** function and a Fair potential regularizer in the following cost function:

$$\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}} \Psi(\boldsymbol{x}), \quad \Psi(\boldsymbol{x}) = \mathbf{1}' h_{.}(\boldsymbol{A}\boldsymbol{x}, \delta_h) + \beta \mathbf{1}' \psi_{.}(\boldsymbol{x}, \delta_\psi),$$

where $\boldsymbol{A}$ is the training data matrix where each row consists of the product of a feature vector and the corresponding label (± 1), $h(\cdot, \delta_h)$ denotes the Huber hinge loss function, and $\psi(\cdot, \delta_\psi)$ denotes the Fair potential, and the dot subscript is the JULIA-style notation. This Fair potential approximates the 1-norm regularizer to encourage the classifier weights $\hat{\boldsymbol{x}}$ to be sparse, in hopes of better generalization performance.

This problem uses the **optimal quadratic majorizer** for the Huber hinge loss from a previous HW problem.

To begin, download the template code `classify-hhinge-fair-how.jl` from Canvas under `Files / homework`.
If you prefer to work in a Jupyter notebook, convert that file using the very helpful utility jupytext at the command line via
```
jupytext -to notebook classify-hhinge-fair-how.jl
```
This utility is useful for many languages, including python.

(a) [5] Add JULIA code to perform the following steps. Submit screenshot of your added code to gradescope.
Apply both your `ncg_inv` and your `ncg_inv_mm` algorithms to minimize the cost function shown above, using $\boldsymbol{x}_0 = \mathbf{0}$.
Use $\delta_h = 0.1$, $\delta_\psi = 0.01$, and $\beta = 80$, with 50 iterations, as shown in the template code.
For `ncg_inv_mm` use the (optimal) Huber curvatures for ψ and use the optimal curvatures from the previous HW problem for the Huber hinge function. Use `ninner=5` inner iterations for the line search for both CG methods.

(b) [5] Make a plot that shows the cost function $\Psi(\boldsymbol{x}_k)$ versus iteration k for both algorithms.
Hint. Your new CG-MM (CG with MM line search) should converge faster than the older CG-GD (CG with GD line search).

(c) [5] Make a figure with two images that compares the classifier feature weights $\hat{\boldsymbol{x}}$ for the LS-based classifier and the Huber hinge classifier.
Hint. You should observe that the latter is sparser.

(d) [5] Make a table that reports the classification accuracies of the nearest subspace classifier, the linear regression classifier, and the Huber hinge classifier for both digits.
Hint. Four of the six numbers needed for the table are already computed for you in the template code. The new Huber hinge classifier should give better accuracy than the subspace and LS-based classifiers.

Non-graded problem(s)

5. [6] **Step size for preconditioned gradient descent**

The Ch. 3 notes derive the following range of step sizes for preconditioned GD (**PGD**):

$$0 < \alpha < \frac{2}{\|\boldsymbol{P}\|_2 L_{\nabla \Psi}},$$

where $\boldsymbol{P}$ is a positive definite preconditioner and $L_{\nabla \Psi}$ denotes the Lipschitz constant of the gradient of the cost function Ψ. This range is valid, but the upper bound is undesirably conservative.

(a) [3] Consider the ordinary LS problem where $\Psi(\boldsymbol{x}) = \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2$ where $\boldsymbol{A}$ has full column rank, and suppose we use the **ideal preconditioner** $\boldsymbol{P} = (\boldsymbol{A}'\boldsymbol{A})^{-1}$. Express the upper bound on the step size above in terms of singular values of $\boldsymbol{A}$.

(b) [3] Determine the ideal step size α_* for this particular preconditioner for the ordinary LS problem?
How does it compare to the upper bound above?

(c) [0] Optional. Find a better (less conservative) expression for the upper bound.

6. [0] **Majorizer properties**

Prove the following additional algebraic properties of **majorizers**.

(a) **Translation property**.
If $g(\boldsymbol{v};\boldsymbol{u})$ is a majorizer for $f(\boldsymbol{v})$, then $\phi_k(\boldsymbol{x}) \triangleq g(\boldsymbol{x} - \boldsymbol{z}; \boldsymbol{x}_k - \boldsymbol{z})$ is a majorizer for $\Psi(\boldsymbol{x}) \triangleq f(\boldsymbol{x} - \boldsymbol{z})$.

(b) **Affine transformation property**.
If $g(\boldsymbol{v};\boldsymbol{u})$ is a majorizer for $f(\boldsymbol{v})$, then $\phi_k(\boldsymbol{x}) \triangleq g(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}; \boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{b})$ is a majorizer for $\Psi(\boldsymbol{x}) \triangleq f(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})$.

(c) If $q_i(t;s)$ is a majorizer for $\psi_i(t)$ then $\phi_k(\boldsymbol{x}) \triangleq \sum_i q([\boldsymbol{B}\boldsymbol{x} - \boldsymbol{b}]_i; [\boldsymbol{B}\boldsymbol{x}_k - \boldsymbol{b}]_i)$ is a majorizer for $\Psi(\boldsymbol{x}) \triangleq \sum_i \psi_i([\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}]_i)$.

(d) Continuing the previous part, if $q_i(t;s) = \psi_i(s) + \dot{\psi}_i(s)(t-s) + \frac{1}{2}c_i(s)(t-s)^2$, then

$$\phi_k(\boldsymbol{x}) \triangleq \Psi(\boldsymbol{x}_k) + \text{real}\{\langle \nabla \Psi(\boldsymbol{x}_k),\, \boldsymbol{x} - \boldsymbol{x}_k \rangle\} + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_k)'\boldsymbol{B}' \, \mathsf{Diag}\{\omega_i([\boldsymbol{B}\boldsymbol{x}_k - \boldsymbol{b}]_i)\} \, \boldsymbol{B}(\boldsymbol{x} - \boldsymbol{x}_k)$$

is a (quadratic) majorizer for $\Psi(\boldsymbol{x}) \triangleq \sum_i \psi_i([\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}]_i)$.