Homework 3 and 4
APPM 5630 Spring 2023
Advanced Convex Optimization

Due date: Friday, Feb 17 2023 at midnight (via Gradescope)
Theme: Convex functions

Instructor: Prof. Becker
Revision date: 2/7/23

Instructions Collaboration with your fellow students is allowed and in fact recommended, although direct copying is not allowed. The internet is allowed for basic tasks. Please write down the names of the students that you worked with. An arbitrary subset of these questions will be graded.

Homework submission instructions at github.com/stephenbeckr/convex-optimization-class/tree/master/Homeworks. You'll turn in a PDF (either scanned handwritten work, or typed, or a combination of both) to **gradescope**, using the link from the Canvas assignment.

Reading Read chapter 3.1, 3.2 and 3.3 in [BV2004]. Students are **strongly advised** to skim appendices A and C in [BV2004] to look for unfamiliar material (and read in more detail if there is unfamiliar material)

Comment For Homeworks 1 and 2, the theoretical portion was meant to be trickier than the coding. In contrast, for this double-set of homeworks, the coding in Homework 4 is likely to be time-intensive, so budget your time accordingly. Each sub-part of HW 4 is progressively weighted less-and-less, so if you don't finish the entire thing, you can still earn a reasonable grade.

Homework 3

Problem 1: Find a 2D function $f(x, y)$ such that $x \mapsto f(x, y)$ is convex for every y, and $y \mapsto f(x, y)$ is convex for every x, but f is not a convex function (that is, it is not jointly convex in (x, y).

Problem 2: [BV2004] Problem 3.14 *Convex-concave functions and saddle-points* (see book for full problem)

Problem 3: [BV2004] Problem 3.16(d): let $f(x_1, x_2) = x_1/x_2$ on $\mathbb{R}^2_{++}$. Determine if this function is convex, concave, quasiconvex and/or quasiconcave. Justify your answers.

Problem 4: [BV2004] Problem 3.18(a): prove that $f(X) = \text{trace}(X^{-1})$ is convex on $\text{dom}\, f = \mathbf{S}^n_{++}$. *Hint: either adapt the proof of concavity of the log-determinant function in §3.1.5 as the book suggests, or use operations-that-preserve-convexity to reduce it to a known convex function.*

Problem 5: [BV2004] Problem 3.36(a). Derive the conjugate of $f(x) = \max_{i=1,\ldots,n} x_i$ on $\mathbb{R}^n$.

Homework 4: deblurring

We will do deblurring of a 1D signal. Given a filter `h = exp(-[-2:2].^2/2)` of length $L = 5$, and a signal x of length $N = 100$, then the discrete circular/periodic convolution $y = h \star x$ is (assuming 1-based indexing, Matlab notation)

$$y[j] = \sum_{i=1}^{L} x[j - i + 1]h[i], \quad j = 1, \ldots, N$$

with the convention that we "wrap" x to make it periodic, e.g., define $x[j] = x[N + j]$ or $x[j] = X[j - N]$ when necessary. (Note: the ugly $+1$ in the formula is due to the 1-based indexing). To perform the circular convolution in Matlab, use either `ifft( fft(x).*fft(h,N), 'symmetric' )` or `cconv(x,h,N)`.

A slightly more realistic model uses a non-circular convolution, e.g., something like Matlab's `conv(_,_,'same')` function, and you are welcome to do that, but it will make the homework a little more complicated (the main issues are extra zero-padding and truncation). Talk to the instructor if you would like to pursue this and need help.

In Python, you can do `numpy.convolve` or, even better (since it will choose a FFT-based algorithm when faster), use `scipy.signal.convolve`. However, these convolutions won't do circular convolutions; that's fine (or even preferable) except that it makes finding the adjoint more difficult, so it's not recommendd. If you wanted more options, then use `scipy.ndimage.convolve` which gives many boundary condition options via the `mode` parameter, and in particular `mode='wrap'` is the circular convolution. But bottom line, it's simpler to implement it yourself with the `fft` and `ifft` commands in the `numpy.fft` package, since otherwise it's more work to find the adjoint (you can also use `rfft` and `irfft` which save a bit of memory and guarantee a real output of `irfft`, but these also complicate the adjoint).

The convolution (whether circular or not) is a linear operator and we'll use it to represent blurring (e.g., from a lens), but it is also used for filtering signals (like tuning a radio) and in machine learning (convolutional neural nets). We will denote the operator as $\mathcal{B}$, and we assume that some blurred and noisy measurements are acquired of a signal x, i.e., $y = \mathcal{B}x + z$ where z is stochastic noise vector. In particular, choose z iid where $z_i \sim \mathcal{N}(0, \sigma^2)$ for standard deviation $\sigma = 0.02$. We will let $x_i = 0$ for all $i = 1, \ldots, 100$ *except* for $x_{10} = 1, x_{13} = -1, x_{50} = 0.3, x_{70} = -0.2$. The filter is $h[j] = e^{-(j-3)^2/2}$ for $j = 1, 2, \ldots, 5$ (this was chosen arbitrarily, but we want everyone to use the same filter for consistency). See Fig. 1.

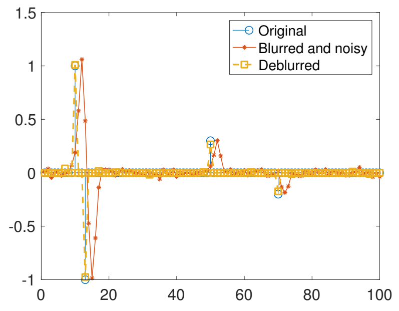

Figure 1: The original signal x and its blurred and noisy version y, and its recovered version. Depending on the exact implementation/definition of your blur, your "blurred and noisy" signal may be shifted left or right a bit; this is OK.

We will estimate x using an optimization model (of course!). Our estimator will be:

$$\hat{x} \in \operatorname*{argmin}_x \|x\|_1 \quad \text{s.t.} \quad \|\mathcal{B}x - y\|_2 \leq \epsilon \tag{1}$$

where we choose $\epsilon = \sigma\sqrt{N}$ since if we evaluate $\|\mathcal{B}x - y\|_2^2$ at the original signal x, this gives $\|z\|_2^2$, which is a χ^2 random variable with mean $\sigma^2 N$; this kind of choice is related to the Morozov discrepancy principle, one of many ways to choose regularization parameters in inverse problems.

Problem 1: [25%] Before we can solve Eq. (1), we need a more explicit representation of $\mathcal{B}$. The blur is a linear operator, so we can build up its matrix representation explicitly by evaluating its output given a complete set of inputs (and the most straightforward choice is to use the standard unit basis as the input).

First, $\boxed{\text{code up a blur function}}$ $\mathcal{B} : \mathbb{R}^N \to \mathbb{R}^N$ defined by $x \mapsto x \star h$; that is, use the fixed filter h, and let x be an input. You may use existing convolution code if you wish (e.g., in Matlab, `cconv(x,h,N)`). This function is known as a **matrix-free** implementation.

Then $\boxed{\text{write a function } \texttt{implicit2explicit}}$ that takes as input a linear function, such as $\mathcal{B}$, and information on the size of the domain (i.e., N), and returns an explicit matrix B such that $\mathcal{B}(x) = B \cdot x$. This function should be coded in a generic way to allow for any linear function $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}^m$ (i.e., returning a $m \times n$ matrix). Demonstrate your function is correct.

Problem 2: [25%] $\boxed{\text{Solve the model (1) using a solver of your choice}}$ (e.g., `cvx` for Matlab, or `cvxpy` for python), and using the explicit matrix B calculated above. Make a single plot showing (1) the original signal x, (2) the blurred and noisy version y, and (3) your estimate $\hat{x}$.

Problem 3: [25%] We will now try to scale this to larger N. As N grows, it becomes disadvantageous to use an explicit matrix B to represent the blur, since this costs $\mathcal{O}(N^2)$ to calculate $B \cdot x$, as well as the time it takes to build B in the first-place. Using fast convolution, $\mathcal{B}(x)$ takes no more than $\mathcal{O}(N \min(L, \log N))$ time. Unfortunately, the very friendly solvers like `cvx` or `cvxpy` do not easily adapt to implicit operators like $\mathcal{B}$, so we will use a first-order method. Before we do that, we will change to a slightly more amenable model. $\boxed{\text{Find a scalar } \lambda}$ such that, for our specific choice of y and ϵ, Eq. 1 gives the same solution as solving

$$\min_x \|x\|_1 + \lambda \|B \cdot x - y\|_2^2. \tag{2}$$

To do this, ask `cvx` or `cvxpy` for a dual variable (that is, a Lagrange multiplier). You should $\boxed{\text{verify}}$, using `cvx` or `cvxpy`, that (1) and (2) really do give the same answer (up to at least 4 or 5 decimal places when you look at the Euclidean norm of the difference). You should still be using the explicit matrix B. Hint: you may need to reformulate the constraint in Eq. (1) to be $\|B \cdot x - y\|_2^2 \le \epsilon^2$.

Problem 4: [15%] Rewrite our model as

$$\min_x \underbrace{\tau \|x\|_1}_{g} + \underbrace{\frac{1}{2}\|\mathcal{B}x - y\|_2^2}_{f}. \tag{3}$$

using $\tau = 1/(2\lambda)$.

First-order solvers will need to know how to compute $f(x)$ and $\nabla f(x) \overset{\text{def}}{=} \mathcal{B}^*(\mathcal{B}x - y)$. Thus for the gradient, we need a function to compute the adjoint $\mathcal{B}^*$. Write a function that computes $\mathcal{B}^*$, and provide evidence that your function is correct. If you would like step-by-step help, try the following steps:

First we will write our own convolution with h. An efficient manner to compute the convolution, and one which is amenable to finding the adjoint, is by writing the convolution in the Fourier domain. Let $\mathcal{F}$ represent the Discrete Fourier Transform (DFT/FFT), with $\mathcal{F}^{-1}$ the inverse Fourier Transform. Note that $\mathcal{F}^* = \mathcal{F}^{-1}$ up to a scaling factor, depending on the convention (which, in the calculation below, will cancel out). Then the circular convolution is $x \mapsto \mathcal{F}^{-1}(\hat{h}. * \mathcal{F}(x))$ where $\hat{h} = \mathcal{F}(h)$ (where we zero-pad h to make it length N), and the ".*" operation represents element-wise multiplication. Writing $\mathcal{H} : z \mapsto \hat{h}. * z$, then the circular convolution is

$$\mathcal{B} = \mathcal{F}^{-1} \circ \mathcal{H} \circ \mathcal{F}.$$

Despite looking complicated, this is actually very useful, as now the adjoint is simple to compute, using the identity $(AB)^* = B^* A^*$:

$$\mathcal{B}^* = \mathcal{F}^* \circ \mathcal{H}^* \circ F^{-*}.$$

For each component (e.g., $\mathcal{H}$), write a function to compute its adjoint, and verify that it is correct (see below). Once this is done, compose all the components together, and you have the adjoint. $\boxed{\text{Turn in code that efficiently computes both } \mathcal{B} \text{ and } \mathcal{B}^*\text{, along with evidence that it correct.}}$

Note: The adjoint of $\mathcal{H} : z \mapsto \hat{h}. * z$ is $\mathcal{H}^* : z \mapsto \bar{\hat{h}}. * z$ where the $\bar{\cdot}$ denotes complex-conjugate.

To provide evidence your adjoint $\mathcal{B}^*$ is correct, here are two possible methods: use your `implicit2explict` method to use $\mathcal{B}^*$ and build B^*, and then verify that B^* really is the adjoint of B (where B is the explicit matrix built from $\mathcal{B}$); or, for several random choices of x and y, verify $\langle \mathcal{B}x, y \rangle = \langle x, \mathcal{B}^* y \rangle$ up to high precision (e.g., 8 decimal places or more). You may wish to write a `test_adjoint` function that automates one of these tests, as this will be useful for future assignments.

Note for Python: `np.allclose` *and* `np.real_if_close` *are useful functions to know about. For dot products, I recommend* `np.vdot` *over* `np.dot` *since it will correctly take the complex conjugate if necessary, and it also plays more nicely arrays of different shapes, such as $(n,)$ or $(n,1)$.*

Problem 5: [10%] Finally, download an existing ℓ_1 first-order solver and solve Eq. (3). In Matlab, recommended packages are the ℓ_1 solvers by Mark Schmidt (e.g., his L1General package, also, his thesis package; in particular, his ℓ_1 solver `L1General2_PSSas` — note that this expects τ to be a vector, not a scalar); and FASTA by Tom Goldstein et al..

In Python, there are many packages, though I don't have first-hand experience to recommend them. What I suggest to use is the `lassoSolver` function that I've defined inside the github.com/stephenbeckr/convex-optimization-class/blob/master/utilities/firstOrderMethods.py file.

Alternatively, in 2018, some students had good success with PyUNLocBoX from Vandergheynst's group at EPFL; see their l1 tutorial. [1] A basic proximal gradient method (last updated 2015) is apgpy (this is written by Brendan O'Donoghue, Stephen Boyd's PhD student and also author of `cvxpy`); some proxes (e.g., for ℓ_1) are in Samuel Vaiter's package pyprox (last updated 2012) and proximity-operator.net; and pyProxSolver by Jiayu Zhou (Asst. Prof at Michigan State).

Solve Eq. (3) with a first-order solver, using the matrix-free operator $\mathcal{B}$; then also solve using the simple solvers from problem 2 (e.g., `cvx` or `cvxpy`) using the matrix B; and verify the solutions are (nearly) the same.

Note: if the solver you use wants a stepsize, use $1/L$ where L is the Lipschitz constant of the gradient of $\frac{1}{2}\|\mathcal{B}x - y\|_2^2$. Recall that $\frac{1}{2}\|\cdot\|_2^2$ has $L = 1$, so the Lipschitz constant of the gradient of $\frac{1}{2}\|\mathcal{B}x - y\|_2^2$ is just $\|\mathcal{B}\|^2$ where $\|\cdot\|$ is the spectral normal. You can compute this on the explict matrix B, or you can compute it on $\mathcal{B}$ using the power method which is pretty simple to implement. There's even a Python implementation on the wikipedia page. Be careful: the standard power method assumes $\mathcal{B}$ is symmetric; since our $\mathcal{B}$ is not symmetric, apply the power method to $\mathcal{B}^*\mathcal{B}$ (which is symmetric/Hermitian) and use $\|\mathcal{B}\|^2 = \|\mathcal{B}^*\mathcal{B}\|$. When you multiply $\mathcal{B}^*\mathcal{B}x$ for an input x, think about where you want the parenthesis: $(\mathcal{B}^*\mathcal{B})x$ or $\mathcal{B}^*(\mathcal{B}x)$?

Problem 6: **[Optional, will not be graded]** Compare the *speed* of the solvers in `cvx`/`cvxpy` with the first-order solver as you increase the size of the problem (generate x and z in any fashion; you can also solve (3) first, for a given λ of your choosing, and then set ε as the norm of the residual).

Problem 7: **[Optional, will not be graded]** Write the code for a 2D blur and its adjoint, and adapt your `implicit2explicit` function to allow for this case. Note: most solvers only work correctly with vector variables, but you can reshape your image into a vector, and just reshape it back to a matrix inside your function. The adjoint of reshape-into-vector is reshape-into-matrix.

You can experiment with different forms of the filter h to recreate motion blurs, out-of-focus blurs, etc.

Problem 8: **[Optional, will not be graded]** For the 1D blur, compare our results with a classical denoising algorithm such as the Lucy-Richardson deconvolution algorithm (in Matlab, this is `deconvlucy` in the Image Processing toolbox; in Python, you could try scikit-image's `skimage.restoration.richardson_lucy`, and see their tutorial).

[1] Their `functions.norm_l2` is $\|\cdot\|_2^2$; to align with our conventions of $\frac{1}{2}\|\cdot\|_2^2$, you'll want to double the value of τ. You'll need to similarly adjust the Lipschitz constant by a factor of 2.