McGill University
Department of
Mathematics and Statistics
Courtney Paquette

McGill

MATH 463/563 - Convex Optimization

Project Description

Due Date: Tuesday, April 11 by 11:59 pm (local Montreal)
Submission Method: CrowdMark for the report and email with subject line `lastname firstname project code` and in the email, a list of all the group members names (first last and student numbers).

1 Project overview

The project for this course involves the implementation of optimization algorithms for image de-blurring and de-noising. This document describes the algorithms that you are required to implement and further guidelines that you are to follow. It also describes a set of problems that you are asked to solve and the expectations of a written report that you are asked to submit with your code. A LaTeX template (`project_template.zip`) has been posted on myCourses for you to use to write your project report.

Rather than submit each of these algorithms as separate pieces of code, you are asked to *combine* all of these algorithms into a single software package. You will also be required to implement more algorithms and options, allowing flexibility for users of your code. All requirements are described in this document.

1.1 Image De-blurring and De-noising

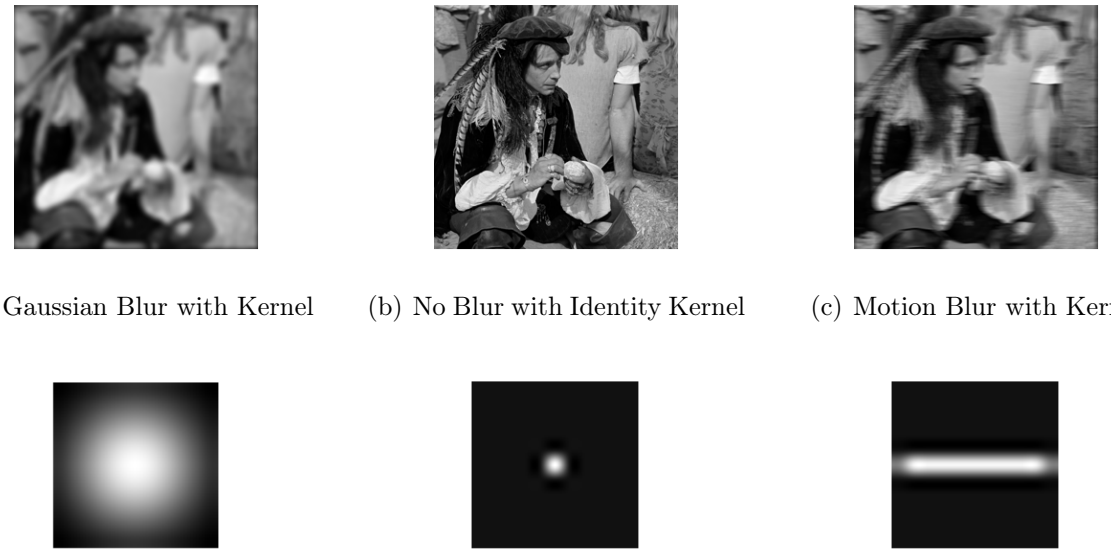
The image de-blurring problem is to recover the original sharp image using a mathematical model fo the blurring process. Many real-life optical systems that cause blurring can be approximated by linear models. Therefore, we use a linear model in our problem too.

We assume that the blurred image is generated from the original image by a convolution with a spatially-invariant kernel (called *point spread function* or *PSF* in the image processing community). The kernel is typically much smaller in size than the image itself. In MATLAB, we use the function `fspecial` and `conv2` for the convolutions. The explicit blurring model we consider is

$$k * x + \eta = b$$

where $*$ represents the 2D-convolution operator, b the corrupted image, k a non-negative blue kernel with relatively small support, x the original *black and white* image (converted to

a matrix with values representing the amount of black color per pixel and *resized so that each pixel is between 0 and 1*), and η additive noise. Three different kernels and the corresponding blurred images are shown in Figure 1.



(a) Gaussian Blur with Kernel (b) No Blur with Identity Kernel (c) Motion Blur with Kernel

Figure 1: Examples of Different Blur Kernels applied to the Man with the Hat Picture

Since convolution is a linear transformation, it can be represented as a matrix multiplication. The matrix representation of convolution, K, is determined from the PSF and the boundary conditions, which describe our assumptions of the scene just outside our image. Modeling the right boundary conditions is critical because different conditions lead to different matrix structures, and so one cannot use the same factorization tricks in all cases. In our case, we assume periodic boundary conditions, that is, the image repeats endlessly in every direction, like tiles on a floor. This prevents boundary artifacts in the blurred image (so the recovered image is also free from boundary artifacts). Another good option would be using zero-padding, but it would work well only if the image had mostly dark pixels near the boundary (like in astronomical images).

With our assumptions, the blurring matrix K turns out to be block circulant with circulant blocks (BCCB). A circulant matrix is diagonalized by the Discrete Fourier Transform (DFT) matrix of the same size, with eigenvalues being the DFT of the first row of the matrix (note that the first row of a circulant matrix gives us all the information about the matrix, so it is not surprising that the eigenvalues of the matrix can be derived from the first row alone). This property is derived directly by writing out the Circular Convolution Theorem in matrix form. This property can be extended to BCCB matrices to show that they have a spectral decomposition $K = Q^H \text{diag}(\lambda) Q$ where $Q = W \otimes W$ is the Kronecker product of the the length-N DFT matrix with itself, and λ is the $2D$ DFT of the convolution kernel.

This structure can be exploited nicely in our problem, which is a specific case of the generic convex optimization problem

$$\min_x \phi_f(Kx - b) + \phi_r(x) + \phi_s(\cdot).$$

with K representing the blurring matrix, $\phi_f(\cdot)$ ensuring fidelity of the recovered image, and $\phi_r(\cdot)$ and $\phi_S(\cdot)$ imposing certain properties on the recovered image. Usually, solving this minimization problem requires solving a system of linear equations with coefficients $I + K^T K$. The size of the matrix K is extremely big, thus making it impractical to try and solve for this system of equations directly. But by the property of BCCB matrices mentioned in the previous paragraph and using the FFT algorithm to calculate the DFT (MATLAB function `fft2`), we can solve this system of equations in $\mathcal{O}(N^2 \log(N))$ operations.

With this set-up, we tackle the non-blind deconvolution problem.

1.1.1 Non-Blind Framework

In the non-blind formulation of the problem, we assume that the kernel is known (for example, a Gaussian or linear kernel as in Figure (1)) and attempt to recover the true image from a blurred, noisy observation. We formulate the image deblurring as the optimization problem

$$\text{minimize}_x \quad \phi_f(Kx - b) + \phi_r(x) + \phi_s(Dx),$$

where ϕ_f, ϕ_r, ϕ_s are convex penalty or indicator functions, K is the blurring operator, b is the blurred image, and D is the discrete gradient operator

$$D = \begin{bmatrix} I \otimes \tilde{D} \\ \tilde{D} \otimes I \end{bmatrix}, \quad \tilde{D} = \begin{bmatrix} -1 & 1 & 0 & \dots & 0 & 0 \\ 0 & -1 & 1 & \dots & 0 & 0 \\ 0 & 0 & -1 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & -1 & 1 \\ 1 & 0 & 0 & \dots & 0 & -1 \end{bmatrix} \in \mathbb{R}^{N \times N}$$

as in [O'Connor and Vandenberghe(2014)]. The first n rows of D are the horizontal discretized derivative operator and the last n rows are the vertical discretized derivative operator, so that minimizing $\phi_s(Dx)$ corresponds to "smoothing" sharp edges in an image and getting rid of noise. Here $x, b \in \mathbb{R}^n$ are vectorized images, and $n = N^2$ is the number of pixels of the image.

We will consider two particular formulations of this problem in this paper. The first, which is valid for any image we wish to deblur, is

$$\underset{x}{\text{minimize}} \quad \|Kx - b\|_1 + \delta_S(x) + \gamma \|Dx\|_{\text{iso}}, \tag{1}$$

where $S = \{x : 0 \leq x \leq \mathbf{1}\}$ is the range of allowed pixel intensities (each image's intensity is normalized to lie within 0 and 1), $\gamma > 0$, and

$$\|(x, y)\|_{\text{iso}} = \sum_{k=1}^{n} (x_k^2 + y_k^2)^{1/2},$$

so that $\gamma\|Dx\|_{\text{iso}} = \gamma\|x\|_{TV}$ gives the total variation norm typically used in the literature. **Here we view x, the image, as a big vector; however when you implement this, you should implement it with x as a matrix.** We choose an l_1 penalty on the residual $Kx - b$ to enforce sparsity because we would like the reconstructed image to be as exact as possible.

The other formulation will be

$$\underset{x}{\text{minimize}} \quad \|Kx - b\|_2^2 + \delta_S(x) + \gamma\|Dx\|_{\text{iso}}. \tag{2}$$

Here: If $x \in \mathbb{R}^n$, then $x \mapsto Dx = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$ where $w_1, w_2 \in \mathbb{R}^n$. Then $\|Dx\|_{\text{iso}} = \|(w_1, w_2)\|_{\text{iso}}$.

1.2 Algorithms

The algorithms we use to solve the non-blind image deblurring problem work for general convex optimization problems of the form:

$$\begin{aligned} \min_{x,y} \quad & f(x) + g(y) \\ \text{subject to} \quad & Ax = y, \end{aligned} \tag{3}$$

where both f and g are closed, convex functions with inexpensive prox-operators. Notice that both the problems we consider can be written in this form, with

$$\begin{aligned} f(x) &= \delta_S(x) \\ g\left([y_1\ y_2]^T\right) = \|y_1 - b\|_1 + \gamma\|(y_2, y_3)\|_{\text{iso}}, \quad A &= \begin{bmatrix} K \\ D \end{bmatrix}, \text{ for Equation (1),} \\ \text{and} \quad g\left([y_1\ y_2]^T\right) = \|y_1 - b\|_2^2 + \gamma\|y_2\|_{\text{iso}}, \quad A &= \begin{bmatrix} K \\ D \end{bmatrix}, \text{ for Equation (2).} \end{aligned}$$

For the moment disregard (3) and consider the problem of finding an x such that

$$0 \in \mathcal{F}(x). \tag{4}$$

Think of $\mathcal{F}$ as $\partial(f + g \circ A)(x)$ so you are trying to find a minimum (in fact, $\mathcal{F}$ is called a maximal monotone operator). We will phrase (3) in terms of different $\mathcal{F}$. The roots of $\mathcal{F}$ precisely coincide with fixed points of the resolvent. Therefore, algorithms to solve for the roots often rely on the fixed point iteration scheme:

$$x^k = (I + t\mathcal{F})^{-1}(x^{k-1}), \quad k = 1, 2, \ldots.$$

This fixed point iteration is only useful in practice when the evaluation of the resolvent is computationally easy. For many scenarios (including image deblurring), this is not the case. Instead, a *splitting algorithm* decomposes the monotone operator as $\mathcal{F} = \mathcal{A} + \mathcal{B}$ where $\mathcal{A}, \mathcal{B}$ are themselves maximal monotone operators whose resolvents are easy to compute.

As described in [Eckstein and Bertsekas(1992)] and [O'Connor and Vandenberghe(2014)], a simple algorithm for solving $0 \in \mathcal{A} + \mathcal{B}$ is the Douglas-Rachford iteration:

$$x^k = (I + t\mathcal{A})^{-1}(z^{k-1}) \tag{5}$$
$$y^k = (I + t\mathcal{B})^{-1}(2x^k - z^{k-1}) \tag{6}$$
$$z^k = z^{k-1} + \rho(y^k - x^k) \tag{7}$$

where t is a positive step size and $\rho \in (0, 2)$ is a relaxation parameter. A derivation of the algorithm described above is obtained by solving for the fixed points of $z^k = F(z^{k-1})$ where

$$F(z) = z + (I + t\mathcal{B})^{-1}(2(I + t\mathcal{A})^{-1}(z) - z) - (I + t\mathcal{A})^{-1}(z).$$

Proven in [Eckstein and Bertsekas(1992)], the sequence z^k converges to a limit of the form $x + tv$ where $0 \in \mathcal{A}(x) + \mathcal{B}(x)$ and $v \in \mathcal{A}(x) \cap -\mathcal{B}(x)$. For more information regarding monotone operators see [Rockafellar and Wets(1998)].

Returning to the problem in (3), optimality conditions are stated in terms of a 0 lying in some subdifferential. For instance, one optimality condition is $0 \in \partial(f + g \circ A)(x) = \mathcal{F}(x)$. It is well known that subdifferentials of closed, convex functions are maximal monotone operators. Therefore, optimizing (3) is analogous to finding a zero of the maximal monotone operator which in turn is the same as finding a fixed point of the resolvent. An example of a monotone operator and its resolvent is the subdifferential and its prox-operator. Unfortunately, the full prox-operator in the image deblurring problem is expensive and so we consider splitting algorithms. Natural choices to "split" the operator $\mathcal{F}(x)$ into $\mathcal{A}(x) + \mathcal{B}(x)$ are the subdifferentials of f and g and/or the subdifferentials of their conjugate duals. Another useful monotone operator is the linear operator $\mathcal{F}(x) = Ax$ where A is skew-symmetric. As the notation suggests, the matrix inverse $(I + \lambda A)^{-1}$ is its resolvent and is explicitly given by

$$\begin{bmatrix} I & \lambda C^T \\ -\lambda C & I \end{bmatrix}^{-1} = \begin{bmatrix} 0 & 0 \\ 0 & I \end{bmatrix} + \begin{bmatrix} I \\ \lambda C \end{bmatrix} (I + \lambda^2 C^T C)^{-1} \begin{bmatrix} I \\ -\lambda C \end{bmatrix}^T \tag{8}$$

In the next few sub-sections, we discuss four different algorithms to solving (3). The first three- Primal Douglas-Rachford, Primal Dual Douglas-Rachford, and ADMM- use the iteration scheme outlined in (5)-(7) under different optimality conditions (maximal monotone operators) and splittings. The last algorithm, Chambolle Pock, approaches the problem from a different perspective. We conclude the introduction with a brief outline of the mathematics behind solving the blind deconvolution problem.

1.2.1 Primal Douglas-Rachford Splitting (Spingarn's Method)

We can rewrite (3) as:

$$\min_{x,y} f(x) + g(y) + \delta_0\big(\begin{bmatrix} A & -I \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \big).$$

The optimality condition for this *primal problem* states that

$$0 \in \partial f(x) + \partial g(y) + \partial\delta_0\big(\begin{bmatrix} A & -I \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \big) =: \mathcal{F}(x, y).$$

Let $\mathcal{A}(x,y) = \partial f(x) + \partial g(y)$ and $\mathcal{B}(x,y) = \partial\delta_0\big(\begin{bmatrix} A & -I \end{bmatrix}(x,y)^T\big)$ so that $\mathcal{F}(x,y) = \mathcal{A}(x,y) + \mathcal{B}(x,y)$. Due to the separability of the variables, the resolvent of $\mathcal{A}$ is $(\text{prox}_{tf}(x), \text{prox}_{tg}(y))$, which we assume is inexpensive to compute. Using the results in the previous section, the operator $\mathcal{B}$'s resolvent is the projection (x,y) of a vector $(\bar{x},\bar{y})$ onto the subspace $\{(x,y) : Ax = y\}$, expressed simply as

$$x = (I + A^TA)^{-1}(\bar{x} + A^T\bar{y}), \quad y = Ax.$$

Although there is a closed form, the inverse leads to implementation difficulties as the size of A is large in the problem we are considering. Applying the iteration scheme (5)-(7), the primal Douglas-Rachford Splitting algorithm is stated below (Algorithm 1).

Algorithm 1: Primal Douglas-Rachford Splitting

Input : Blurred image b, step size t, relaxation parameter ρ
Output : Deblurred image x
Initialize : (z_1^0, z_2^0)

for $k = 1, 2, \ldots,$ **do**
  $x^k \leftarrow \text{prox}_{tf}(z_1^{k-1})$; `// Performs resolvent of` $\mathcal{A}$
  $y^k \leftarrow \text{prox}_{tg}(z_2^{k-1})$;
  $u^k \leftarrow (I + A^TA)^{-1}\big(2x^k - z_1^{k-1} + A^T(2y^k - z_2^{k-1})\big)$; `// Performs resolvent of` $\mathcal{B}$
  $v^k \leftarrow A(u^k)$;
  $z_1^k \leftarrow z_1^{k-1} + \rho(u^k - x^k)$;
  $z_2^k \leftarrow z_2^{k-1} + \rho(v^k - y^k)$;
end
return $x\text{Sol} = \text{prox}_{tf}(z^k)$;

1.2.2 Primal-Dual Douglas Rachford Splitting

Consider the dual problem of (3):

$$\text{maximize}_y \quad -f^*(-A^Ty) - g^*(y) \tag{9}$$

The optimality conditions for both the primal (3) and the dual (9) can be written as

$$0 \in \begin{bmatrix} 0 & 0 & A^T & I \\ 0 & 0 & -I & 0 \\ -A & I & 0 & 0 \\ -I & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} + \begin{bmatrix} 0 \\ \partial g(y) \\ 0 \\ \partial f^*(w) \end{bmatrix}$$

where w is the dual variable and z is the dual multiplier for the constraint $y = Ax$. This can be reduced to the smaller system

$$0 \in \begin{bmatrix} 0 & A^T \\ -A & 0 \end{bmatrix} \begin{bmatrix} x \\ z \end{bmatrix} + \begin{bmatrix} \partial f(x) \\ \partial g^*(z) \end{bmatrix},$$

by using the relationships $\partial g^* = (\partial g)^{-1}$ and $\partial f^* = (\partial f)^{-1}$ and eliminating variables. The right hand side is now a sum of two monotone operators in x and z:

$$\mathcal{A}(x,z) = \begin{bmatrix} \partial f(x) \\ \partial g^*(z) \end{bmatrix}, \quad \mathcal{B}(x,z) = \begin{bmatrix} 0 & A^T \\ -A & 0 \end{bmatrix} \begin{bmatrix} x \\ z \end{bmatrix},$$

which we will use as the "split" for the monotone operator. Unlike the "split" in the Primal Douglas-Rachford algorithm, this uses both optimality conditions of the primal problem *and* the dual problem. Applying the iteration scheme (5-7), the primal Douglas-Rachford Splitting algorithm is stated below (Algorithm 2).

Algorithm 2: Primal-Dual Douglas-Rachford Splitting

Input : Blurred image b, step size t, relaxation parameter ρ
Output : Deblurred image x
Initialize : p^0, q^0

for $k = 1, 2, \ldots,$ **do**
- $x^k \leftarrow \text{prox}_{tf}(p_1^{k-1})$; `// Performs resolvent of` $\mathcal{A}$
- $z^k \leftarrow \text{prox}_{tg^*}(q^{k-1})$;
- $\begin{bmatrix} w^k \\ v^k \end{bmatrix} = \begin{bmatrix} I & tA^T \\ -tA & I \end{bmatrix}^{-1} \begin{bmatrix} 2x^k - p^{k-1} \\ 2z^k - p^{k-1} \end{bmatrix}$; `// Performs resolvent of` $\mathcal{B}$
- $p^k \leftarrow p^{k-1} + \rho(w^k - x^k)$;
- $q^k \leftarrow q^{k-1} + \rho(v^k - z^k)$;

end

return $x\text{Sol} = \text{prox}_{tf}(p^k)$;

1.2.3 ADMM (Dual Douglas-Rachford)

It can be shown that the alternating direction method of multipliers (ADMM) is exactly the Douglas-Rachford algorithm applied to the dual problem (9), which we rewrite as

$$\text{minimize}_z \quad f^*(-A^T z) + g^*(z) := h(z) + k(z). \tag{10}$$

The optimality conditions for (10) are precisely $0 \in \partial h(z) + \partial k(z)$. This is a sum of two maximal monotone operators $\mathcal{A}(z) = \partial h(z)$ and $\mathcal{B}(z) = \partial k(z)$, with resolvents $\text{prox}_{th}(z)$ and $\text{prox}_{tk}(z)$, respectively. An equivalent formulation of the Douglas-Rachford iteration for this problem, obtained by simply interchanging the order of the updates and performing a change of variables, is thus:

$$v^k = \text{prox}_{th}(z^{k-1} - w^{k-1}) \tag{11}$$
$$z^k = \text{prox}_{tk}(v^k + w^{k-1}) \tag{12}$$
$$w^k = w^{k-1} + \rho v^k + (1-\rho) z^{k-1} - z^k. \tag{13}$$

Proposition 1.1. *If $h(z) = f^*(-A^T z)$, then $prox_{th}(z)$ can be computed by the following:*

$$\hat{x} = \arg\min_x \left(f(x) + z^T Ax + \frac{t}{2}\|Ax\|^2 \right)$$

$$prox_{th}(z) = z + tA\hat{x}.$$

Proof. Notice that $\hat{x}$ as given above is optimal for

$$\begin{aligned} \text{minimize} \quad & f(x) + \frac{t}{2}\|w\|_2^2 \\ \text{subject to} \quad & w = Ax + z/t, \end{aligned}$$

where z is fixed and x, w are the variables. The optimality conditions for the above are $A\hat{x} + z/t = \hat{w}$, $-A^T\hat{u} \in \partial f(\hat{x})$, and $t\hat{w} = \hat{u}$, where u is the dual multiplier. Eliminating x and w and using the relationship $(\partial f)^{-1} = \partial f^*$, this is equivalent to the condition that $0 \in -A\partial f^*(-A^T\hat{u}) + \frac{1}{t}(\hat{u} - z)$, which is precisely the optimality condition that guarantees $\hat{u} = \text{prox}_{th}(z) = \arg\min_u f^*(-A^T u) + \frac{t}{2}\|u - z\|_2^2$. Combining these optimality conditions also leads to the relationship $\hat{u} = z + tA\hat{x}$, which proves the proposition. □

By this proposition (applied to both h and k in (10)) the Douglas-Rachford algorithm on the dual can be rewritten as

$$\begin{aligned} \hat{x}^k &= \arg\min_x \left(f(x) + (z^{k-1} - w^{k-1})^T Ax + \frac{t}{2}\|Ax\|^2 \right) \\ v^k &= z^{k-1} - w^{k-1} + tA\hat{x}^k \\ \hat{y}^k &= \arg\min_y \left(g(y) - (v^k + w^{k-1})^T y + \frac{t}{2}\|y\|^2 \right) \\ z^k &= v^k + w^{k-1} - t\hat{y}^k \\ w^k &= w^{k-1} + \rho v^k + (1-\rho)z^{k-1} - z^k. \end{aligned}$$

If $\rho = 1$, the w update simplifies to $w^k = t\hat{y}^k$ and the entire algorithm simplifies to

$$\hat{x}^k = \arg\min_x L(x, \hat{y}^{k-1}) \tag{14}$$

$$\hat{y}^k = \arg\min_y L(\hat{x}^k, y) \tag{15}$$

$$z^k = z^{k-1} + t(A\hat{x}^k - \hat{y}^k), \tag{16}$$

where $L(x, y, z) = f(x) + g(y) + z^T(Ax - y) + \frac{t}{2}\|Ax - y\|_2^2$ is the augmented Lagrangian for the primal problem. This is the formulation of ADMM which we saw in class. To obtain the overrelaxed version for $0 < \rho < 2$, we simply replace instances of $A\hat{x}^k$ in (15) and (16) with $\rho A\hat{x}^k + (1-\rho)\hat{y}^{k-1}$.

In our implementation of ADMM for the particular problem (3), we first introduce a new variable u and add the constraint that $x = u$. We can then rewrite the general problem as

$$\text{minimize} \quad f(u) + g(y)$$

$$\text{subject to} \quad \begin{bmatrix} I \\ A \end{bmatrix} x + \begin{bmatrix} -I & 0 \\ 0 & -I \end{bmatrix} \begin{bmatrix} u \\ y \end{bmatrix} = 0.$$

The augmented Lagrangian for this problem is

$$L(x, u, y, w, z) = f(u) + g(y) + w^T(x - u) + z^T(Ax - y) + \frac{t}{2}\left(\|x - u\|^2 + \|Ax - y\|^2\right).$$

We now alternate between minimizing L over x and minimizing L over (u, y). Since the problem is separable in u and y, this actually amounts to three separate minimization problems. Straightforward computations then show that the (overrelaxed) algorithm is as below (Algorithm 3).

Algorithm 3: Alternating Direction Method of Multipliers (ADMM)

Input : Blurred image b, step size t, relaxation parameter ρ
Output : Deblurred image x
Initialize : x^0, u^0, y^0, w^0, z^0

for $k = 1, 2, \ldots,$ **do**
- $x^k \leftarrow (I + A^TA)^{-1}\left(u^{k-1} + A^Ty^{k-1} - \frac{1}{t}(w^{k-1} + A^Tz^{k-1})\right)$;
- $u^k \leftarrow \text{prox}_{t^{-1}f}(\rho x^k + (1-\rho)u^{k-1} + w^{k-1}/t)$;
- $y^k \leftarrow \text{prox}_{t^{-1}g}(\rho Ax^k + (1-\rho)y^{k-1} + z^{k-1}/t)$;
- $w^k \leftarrow w^{k-1} + t(x^k - u^k)$;
- $z^k \leftarrow z^{k-1} + t(Ax^k - y^k)$;

end

return $x\text{Sol} = (I + A^TA)^{-1}\left(u^k + A^Ty^k - \frac{1}{t}(w^k + A^Tz^k)\right)$;

1.2.4 Chambolle-Pock Method (Honor students only)

As described in [Chambolle and Pock(2011)], the Chambolle-Pock method solves the generic convex optimization problem in 3 by finding the solution $(\hat{x}, \hat{y})$ of the saddle point problem

$$\min_x \max_y \langle Ax, y\rangle + f(x) - g^*(y). \tag{17}$$

In our case, this turns out to be the Lagrangian of the general problem (3), with $f(x)$, $g(x)$ and A as defined there. The algorithm is described below in Algorithm (4).

By choosing the step sizes s and t such that $st(\|A\|_2^2) < 1$, we can ensure the following convergence:

$$\frac{\|y^k - \hat{y}\|_2^2}{2s} + \frac{\|x^k - \hat{x}\|_2^2}{2t} \le C\left(\frac{\|y^0 - \hat{y}\|_2^2}{2s} + \frac{\|x^0 - \hat{x}\|_2^2}{2t}\right). \tag{18}$$

This establishes that the sequence of iterates, (x^k, y^k), is bounded. Therefore one can find a subsequence (x^{k_m}, y^{k_m}) that converges to some limit (x^*, y^*). It can also be shown that x^{k_m-1} and y^{k_m-1} also converge respectively to x^* and y^*. This shows that (x^*, y^*), obtained

Algorithm 4: Chambolle-Pock Algorithm

Input : Blurred image b, step sizes t, s
Output : Deblurred image x
Initialize : x^0, y^0, z^0

for $k = 1, 2, \ldots,$ **do**
  $y^k = \text{prox}_{sg^*}(y^{k-1} + sAz^{k-1})$;
  $x^k = \text{prox}_{tf}(x^{k-1} - tA^T y^k)$;
  $z^k = 2x^k - x^{k-1}$;
end
return $x\text{Sol} = x^k$;

as a limit of the above algorithm, is a fixed point of the iterations, and hence a saddle point of the min-max problem (17).

There are a couple of important differences between the Chambolle-Pock algorithm and the previous variants of the Douglas-Rachford algorithm. One, the step sizes s and t in Chambolle-Pock are dependent on the size of the operators in the problem (i.e. the maximum singular value of matrix A). Second, this algorithm does not require solving a system of linear equations involving $I + A^T A$. We only need vector-matrix multiplications of A and A^T.

Despite these operational differences, at its core, the Chambolle-Pock algorithm can be interpreted as a form of preconditioned ADMM. The first step of ADMM solves a least squares problem, which involves a difficult matrix inversion. In general, this could be quite an expensive operation. To get around this, the Chambolle-Pock algorithm adds

$$\frac{1}{2}\langle\left(\frac{1}{s} - tAA^T\right)(y - y^{k-1}), y - y^{k-1}\rangle$$

to the augmented Lagrangian to be minimized at the k-th iteration:

$$y^k = \arg\min_y g^*(y) - \langle A^T y, x^{k-1}\rangle + \frac{t}{2}\|A^T y + z^{k-1}\| + \frac{1}{2}\langle(\frac{1}{s} - tAA^T)(y - y^{k-1}), y - y^{k-1}\rangle$$

This simplifies to

$$y^k = (I + s\partial g^*)^{-1}(y^{k-1} + sAz^{k-1})$$

where

$$z^k = 2x^k - x^{k-1},$$

which are exactly two of the steps of the algorithm described above.

2 Coding guidelines

Upon completion of your Matlab software package, a user should be able to run any of your implemented algorithms with various sets of inputs using the following command:

```
» x = packagename('problem', 'algorithm',x, kernel, b,i);
```

Here, `packagename` is the name of your software. You can call it something descriptive, such as `optsolver`, or you can be more creative. Below is a description of the inputs. The input/output `x` should be the initial iterate to start the algorithm and final iterate (de-blurred and de-noised image), the latter of which will hopefully be a stationary point, or even a local minimizer, the `b` is the blurred and noisy image (see below on how to generate it), the `'algorithm'` is the algorithm you are running, and `kernel` is the convolution kernel used to blur the image (see below).

- `'problem'`: A problem will be specified by either '`l1`' or '`l2`' indicating if we are using (1) or (2) respectively.
- `'algorithm'`: The algorithm to be run will be specified by the user through this input. You should structure your code so that the following options are valid:
  - `douglasrachfordprimal`, Primal Douglas-Rachford Splitting, Algorithm 1;
  - `douglasrachfordprimaldual`, Douglas-Rachford primal-dual algorithm, Algorithm 2;
  - `admm`, Dual Douglas-Rachford algorithm, Algorithm 3;
  - `chambollepock`, Chambolle-Pock Method, Algorithm 4; honor students only
- `kernel`: The kernel, k, used to do the convolution. See `fspecial` below.
- `b`: The blurred and noisy image. See below on how to generate it
- `i`: Input parameter values will be specified by the user through the *structure* `i`. You create a structure in Matlab whenever you define a variable with a dot (i.e., `.`) in its name. For example, your code *must* allow the user to specify the maximum number of iterations (call it `maxiter`) and amount of de-noising γ in (1)/(2). Then, a particular call to your program may be the following:

```
» i.maxiter = 500;
» i.gammal1 = 0.049;
» i.tprimaldr = 2.0;
» i.rhoprimaldr = 0.1;
» i.tprimaldualdr = 2.0;
» i.rhoprimaldualdr = 1.049;
» x = optsolve('l1', 'douglasrachfordprimal', x, kernel, b, i);
» x = optsolve('l1','douglasrachfordprimaldual', x, kernel, b, i);
```

  Note that the structure `i` includes both parameters, and so both can be passed through your code simply by passing `i`. A complete list of parameters that you are required to allow the user to specify is the following. You are allowed to define more parameters, if you wish, but at least all of these are necessary. Any values that the user does not specify should be set to a default value of your choice. (In other words, a user should not be forced to specify any of these values in order to run your code. If they choose not to set a value, then your code must still run—with the default values that you have specified.)

- `maxiter`: iteration limit, i.e., the value such that if k is more than `i.maxiter`, then your code should terminate as the iteration limit has been reached;
- `gammal1`, `gammal2`: amount of de-noising you want to have (see (1) and (2)), respectively; you should try various values to see what works best
- `rhoprimaldr`, `tprimaldr`: relaxation parameter ρ and stepsize t in primal Douglas-Rachford splitting algorithm (Algorithm 1)
- `rhoprimaldualdr`, `tprimaldualdr`: relaxation parameter ρ and stepsize t in primal-dual Douglas-Rachford splitting algorithm (Algorithm 2) `rhoadmm`, `tadmm`: relaxation parameter ρ and stepsize t in ADMM algorithm (Algorithm 3)
- `tcp`, `scp`: step size parameters for the Chambolle-Pock Algorithm; honor students only

Preferably, *every* constant in your code should be an input that the user can specify, if they so choose.

As the intent of this project is for you to create a software package, you should not repeat code. In particular, it is **not** acceptable if you create an `if` statement in your main program and write separate code for each algorithm, such as in the following (i.e., **the following is bad code!**):

```
if strcmp('algorithm','douglasrachfordprimal')==1

 // all the code for Primal Douglas-Rachford Algorithm...

elseif strcmp('algorithm','admm')==1

  // all the code for ADMM...

elseif ...
```

The problem with coding like this is that you'll have to repeat certain procedures many times, which often leads to errors (bugs); e.g., every case above is going to require a termination check, even though the same termination check should be used for each algorithm. Thus, instead of structuring your main program in this manner, you should structure it so that no matter which algorithm is being run, all calls will follow the same series of steps. Indeed, it would be preferable to structure the main program as a series of generic steps (similar to pseudocode), with all of the details relegated to subroutines, for example, reuse proximal operators such as prox on ℓ_1:

» `boxProx`(x): computes the projection onto $0 \leq x \leq 1$
» `l1Prox`(x): computes the proximal operator of the ℓ_1-norm, $\|\boldsymbol{x}\|_1$
» `isoProx`(x, γ): computes the proximal operator of the $\gamma\|(x_1, x_2)\|_{\text{iso}}$.

A few more comments and guidelines:

- The command in MATLAB, `I = imread('imagename.png')`, stores the 'imagename.png' as a matrix of size number of pixels $\times$ number of pixels. To convert the image to black and white, you can use the command, `rgb2gray(I)` to convert the image to black and white.

- To resize the image so that each pixel is between 0 and 1, one can use the following code:

```
I = double(I(:, :, 1));
mn = min(I(:));
I = I - mn;
mx = max(I(:));
I = I/mx;
```

- For displaying an image, one can use the following:

```
figure('Name','image before deblurring')
imshow(I,[])
```

  The 'Name' and 'image before deblurring' in `figure` names the image 'image before deblurring'.

- For adding convolution kernels and noise η to an image, MATLAB has some pre-built code. One can use `fspecial`, for blurring (see `https://www.mathworks.com/help/images/ref/fspecial.html`), and `imnoise` for adding different types of noise (see `https://www.mathworks.com/help/images/ref/imnoise.html`). You should try various different noise and convolution kernels (at least 2 different kernels and 2 different types of noises with various intensities). Record your results in the report. Given a kernel (see below), you can perform $k * I$ by

```
b = imfilter(I,kernel);
```

  To add noise, you can use

```
b = imnoise(b,'noisetype');
```

  The resulting `b` will have `b` = kernel $* I +$ 'noisetyle'. This will be your b in (1) and (2).

  You may also need to resize the image in order to make the algorithms run faster. To do this, you can use

```
» resizefactor = 0.1;
» I = imresize(I, resizefactor);
```

  Examples of different kernels and types of noise you should try and record in your report (please try even more!):

  - Gaussian kernel: `kernel = fspecial('gaussian',hsize,sigma)` returns a rotationally symmetric Gaussian lowpass filter of size `hsize` with standard deviation `sigma`. The `hsize` should be significantly smaller than the size of the image, like $[15, 15]$.

- Motion kernel: `fspecial('motion',len,theta)` returns a filter to approximate, once convolved with an image, the linear motion of a camera. len specifies the length of the motion and theta specifies the angle of motion in degrees in a counter-clockwise direction. The filter becomes a vector for horizontal and vertical motions. The default len is 9 and the default theta is 0, which corresponds to a horizontal motion of nine pixels.
- Salt-and-Pepper noise with noise density: `b = imnoise(b,'salt & pepper',d)` adds salt and pepper noise, where d is the noise density. This affects approximately d*numel(I) pixels. Try for instance, $d = 0.5$
- Gaussian noise: `b =imnoise(b,'gaussian',m,vargauss)` adds Gaussian white noise with mean m and variance vargauss.

Sample code to generate a blurred image from an `'image.png'`

```
» I = imread('image.png');
» I = rgb2gray(I);   %black and white
» I = double(I(:, :, 1));
» mn = min(I(:));
» I = I - mn;
» mx = max(I(:));
» I = I/mx;
» figure('Name','image before blurring')
» imshow(I,[])
» kernel = fspecial('gaussian', [15,15], 5);
» b = imfilter(I,kernel);
» b = imnoise(b,'salt & pepper',noiseDensity);
```

- Please see `multiplyingMatrix.m` for fast computations of computing $k * x$ and Dx. The trick is to use fast fourier computations. Without this, you may not be able to compute large matrix multiplications on your computer.

- It is a strict requirement that **your code must be well-commented**. A good rule of thumb is that I should be able to follow all of the steps in your code without looking at any of the code at all! I should be able to follow everything simply by reading the in-line comments.

- You are encouraged to augment the output produced by the code. You should print during every iteration (iteration number, objective value, etc) but you may decide to print more information. **One thing that you are required to add to your code is a summary to be printed at the end of the output.** The summary should include a statement of the result (i.e., whether the problem was solved, the iteration limit was reached, etc.), the final objective value, and the CPU time. Matlab reveals CPU time through the `tic` and `toc` commands. Place `tic` as the first line of your main program. This will start the clock. Then, whenever you invoke the command `toc`, its value will be the time (in seconds) elapsed since `tic` was called.

Test problems

I have provided a few images for you to test on and they are posted on myCourses (see `testimages.zip`). You should use these images as test problems to check that your code is working correctly, but it is also a wise idea to create your own test problems so that your code solves more than these problems!

Project report

You are required to submit a written report with your software containing the following:

- Report should not exceed 15 pages (you may put some figures in an Appendix which does not count against your page limit total; however the 15 pages should be a complete report (i.e. it should contain figures and reflect what you want the reader to learn from your experiments). On **page 16**, include a list of each group member (full name and student idea) with a paragraph of exactly what each person did.
- Summarize each algorithm in a few sentences each. Your descriptions should include a short derivation and how the algorithms differ from each other.
- Summarize the problem that you are solving (image deblurring and denoising)
- Summarize all the proximal operators that you are using and their explicit derivations
- Provide a table of default input parameters for your code, which may or may not vary between algorithms. (The iteration limit should be `500`, but you are asked to provide default values for the remaining parameters.) From testing your code, you should pick values that you believe to be the best for your algorithms. You should describe and show (via graphs or images) the effect of the different parameters
- You should experiment with all the different parameters: γ, different kernels, types of noise, effect of ρ and step sizes.
- Provide visual interpretations (*e.g.* graph of the function values relative to the iterations, etc) of the results showing the performance of all the algorithms. Outputs of different images under different convolution kernels and noise levels at the begining and end for the various algorithms and two objective functions.
- Comment on the results. Was any algorithm consistently the best? If not, can you guess why some algorithms had trouble with certain problems?
- Summarize your experience with this project. Would you declare any of your algorithms the winner? Consideration for that distinction should include the algorithm's performance, but also how easy it was to code and how many parameters you needed to "tune" before it worked well. Indeed, you should comment on your experience coding all of the algorithms and describe your impressions of each. What method would you recommend to an expert coder? What method would you recommend to a user who is not an expert in coding or in convex optimization? Which of the two objective functions seemed to work better? Did this depend on the type of noise or kernel?

Project grading

Do not be discouraged if you cannot get all of your algorithms to solve all problems. **It is better to code some of the algorithms correctly than to code all of them poorly.**

Your grade for the project will be based on the merits of your (well-commented!) code as well as the clarity of presentation in your report. So that I can easily run your code with the default parameters that you have specified, you are required to provide a file that will run all of your algorithms with the default parameters for a specific problem and a specific starting point. In particular, this file should be structured in the following way,

```
% Set common input parameters for all algorithms
i.maxiter = 500;
i.gammal1 = 0.049;

% Set default input parameters for primal douglas-rachford algorithm
i.tprimaldr = 2.0;
i.rhoprimaldr = 0.1;

% Run Douglas-Rachford Primal Algorithm
» x = optsolve('l1', 'douglasrachfordprimal', x, kernel, b, i);

% Set default input parameters for Douglas-Rachford Primal-Dual Algorithm...
```

Note that I get to choose the starting point when I run your code!

In addition to the provided images, I also plan to run everyone's code on a "secret" set of images that I will not provide (at least, not until all projects have been submitted).

Please ask me if you have any questions about my expectations or anything else!

(Finally, note that you are expected to work on the code and report for this project *in your own group*. If there is any evidence of sharing code or writing in your report, beyond your group, then there will be *serious* unfortunate consequences.)

References

[Chambolle and Pock(2011)] Antonin Chambolle and Thomas Pock. A first-order primal-dual algorithm for convex problems with applications to imaging. *J. Math. Imaging Vision*, 40(1): 120–145, 2011. ISSN 0924-9907. doi: 10.1007/s10851-010-0251-1. URL `https://doi-org.proxy3.library.mcgill.ca/10.1007/s10851-010-0251-1`.

[Eckstein and Bertsekas(1992)] Jonathan Eckstein and Dimitri Bertsekas. On the douglas-rachford splitting method and the proximal point algorithm for maximal monotone operators. *Mathematical Programming*, 55(3):293–318, 1992.

[O'Connor and Vandenberghe(2014)] Daniel O'Connor and Lieven Vandenberghe. Primal Dual Decomposition by Operator Splitting and Applications to Image Deblurring. *Siam Journal Imaging Sciences*, 7(3), 2014.

[Rockafellar and Wets(1998)] R.T. Rockafellar and R.J.-B. Wets. *Variational Analysis*. Springer, Berlin, 1998.