# Computational Imaging Laboratory: Part I: MAP Restoration with Gaussian Prior

October 16, 2025

## 1 Introduction

This laboratory explores the use of maximum *a posteriori* (MAP) estimation of images from noisy and blurred data using a Gaussian Markov random field (MRF) prior model. This estimation methodology is widely useful for problems of image restoration and reconstruction because it can be used to explicitly model both measurement noise and prior knowledge regarding the image to be estimated.

In order to use a MAP estimation framework, we will need three major components:

- *Prior model* - This is the probability distribution that we assume for the unknown image to be estimated. We will use and MRF prior model because it captures essential characteristics of images while remaining computationally tractable.
- *Measurement noise model* - We introduce a general model for the measurement which consists of a linear blurring operator followed by additive Gaussian noise.
- *Optimization method* - The MAP estimation framework results in a high dimensional optimization problem that must be solved. There are many ways to solve this problem, but we introduce a useful optimization technique known as iterative coordinate descent (ICD).

**All solutions for this laboratory should be implemented using the JAX package in Python.** It is important that students learn to program in JAX because it is efficient and highly portable, but it requires some experience to construct a high quality program that can be easily debugged.

### 1.1 Preparation for JAX

In this lab, you are required to use JAX. To help you get started, the GitHub repository `jax_startup` offers useful resources for setup. You can retrieve the `jax_startup` repository by running the following command:

```
git clone git@github.com:cabouman/jax_startup
```

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Your first task is to install the necessary packages for this lab. You can follow the instructions in the README of the `jax_startup` repository to set up the conda environment and install the required packages.

It is also strongly recommended that you run and study the FIR demo provided in the repository. You can run the demo by following the instructions in the README. In the demo, you will learn how to use the `jax.vmap` function, which is essential for completing this lab.

## 2 Prior Model Formulation

For the purposes of this laboratory, the random field $\{X_s\}_{s\in S}$ denotes an image with continuously valued pixels taking values on the lattice $S$ with $N$ points. Furthermore, $X$ is assumed to be an MRF with a strictly positive density; so by the Hammersly-Clifford theorem, its density must have the form of a Gibbs distribution. We also make the common assumption that the Gibbs distribution for $X$ uses only pairwise interactions. In this case, the set of all cliques is given by

$$\mathcal{C} = \{\{s, r\} | \, r \in \partial s \text{ for } s, r \in S\} \ ,$$

where $\partial j$ denotes the neighbors of $j$. For this laboratory, we use an 8-point neighborhood system.

In many practical applications, the Gibbs density of the MRF is assume to have the following form

$$p(x) = \frac{1}{Z}\exp\left\{-\sum_{\{s,r\}\in\mathcal{C}} b_{s,r}\rho(x_s - x_r)\right\} \ , \tag{1}$$

where $\rho(\cdot)$ is a real valued, positive and symmetric function and $z$ is the normalizing constant known as the partition function. There are a wide variety of common choices for this function, but for this portion of the laboratory we focus on a class of MRF's known as generalized Gaussian MRF's with the form

$$p(x) = \frac{1}{Z}\exp\left\{-\frac{1}{p\sigma_x^p}\sum_{\{s,r\}\in\mathcal{C}} g_{s,r}\,|x_s - x_r|^p\right\} \ , \tag{2}$$

$$\tag{3}$$

where $g_{s,r}$ are non-causal prediction coefficients, $\sigma_s^2$ is the non-causal prediction variance, and $z$ is the normalizing constant for the distribution. We assume that the function $g_{s,r}$ is normalized so that

$$1 = \sum_{r\in\partial s} g_{s,r} \ . \tag{4}$$

For simplicity, we assume that $S$ is a rectangular lattice, that the function $g$ is shift invariant so that $g_{s,r} = g_{s-r} = g_{r-s}$, and we use the "clamp" boundary condition that is the default in JAX. With these assumptions, (4) holds whenever $1 = \sum_s g_s$. In this case, it can be shown that

$$\begin{aligned} p(x_s|x_r \, r \neq s) &= p(x_s|x_r, r \in \partial s) && (5)\\ &= \frac{1}{Z}\exp\left\{-\frac{1}{p\sigma_x^p}\sum_{r\in\partial s} g_{s-r}\,|x_s - x_r|^p\right\} \ . && (6)\end{aligned}$$

For the special case of $p = 2$, the GGMRF is Gaussian. In this case, the distribution can be written in matrix form as

$$p(x) = \frac{1}{(2\pi)^{(N/2)}}|B|^{1/2}\exp\left\{-\frac{1}{2}x^t B x\right\} \tag{7}$$

where $x$ is a vectorized form of the image and $B$ is a symmetric positive definite matrix with entries

$$B_{s,r} = \frac{1}{\sigma_x^2} \left( \delta_{s-r} - g_{s,r} \right) \ .$$

For a GMRF, the conditional density of a pixel is given by

$$p(x_s | x_r \, r \neq s) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp \left\{ -\frac{1}{2\sigma^2} \left( x_s - \sum_{r \in \partial s} g_{s-r} x_r \right)^2 \right\} \ . \tag{8}$$

So we can see that the conditional expectation and variance are given by

$$\begin{aligned} E\left[ x_s | x_r, r \neq s \right] &= \sum_{r \in \partial s} g_{s-r} x_r & (9) \\ Var\left[ x_s | x_r, r \neq s \right] &= \sigma^2 \ . & (10) \end{aligned}$$

## Section Problems:

**Instructions:**
Do the following problems using JAX and/or numpy.

- Model the image $X$ as a GGMRF clamp boundary conditions with parameters $g_i$ given as shown below.

| | | |
|---|---|---|
| 1/12 | 1/6 | 1/12 |
| 1/6 | 0 | 1/6 |
| 1/12 | 1/6 | 1/12 |

(11)

- Use the image img04.tif for your experiments, but convert it to floats and scale it owith pixel values in the range $[0, 255]$.

**Problems:**

1. Read in image data, $x$, to be used for your experiments by performing the following operations:
   (a) Read the file img04.tif that takes values in the range $[0, 255]$
   (b) Convert the data to floats.
   (c) Scale the data to the range $[0, 1]$ by multiplying by the value $1/255$.
2. Compute the noncausal prediction error for the image img04.tif

$$e_s = x_s - \sum_{r \in \partial s} g_{s-r} x_r$$

   and display it as an image by adding an offset of 0.5 to each pixel and clipping values to the range $[0, 1]$.

## 3 MAP Denoising with Gaussian Prior

In this section, you will use a Gaussian MRF to compute the MAP estimate of an image that has been corrupted using additive Gaussian noise. Assume that the noisy image Y is related to the noiseless image X by

$$Y = X + W \ ,$$

where $W \sim N(0, \sigma^2)$ is additive white Gaussian noise (AWGN) with mean 0 and variance $\sigma^2$. Furthermore, assume that X is a Gaussian MRF with noncausal prediction variance of $\sigma_x^2$ and a noncausal prediction filter of $g_i$ shown in (11).

The MAP estimate is given by

$$\begin{aligned} \hat{x} &= \arg\max_x \left\{ p_{x|y}(x|y) \right\} \\ &= \arg\max_x \left\{ \log p(y|x) + \log p(x) \right\} \ , \end{aligned}$$

where

$$\begin{aligned} p(y|x) &= \frac{1}{(2\pi\sigma^2)^{N/2}} \exp\left\{ -\frac{1}{2\sigma^2} \|y - x\|^2 \right\} \\ p(x) &= \frac{1}{z} \exp\left\{ -\frac{1}{2} x^t B x \right\} \ . \end{aligned}$$

where $B = R^{-1}$ is the precision matrix for $X$. Therefore, the MAP estimate of $x$ is given by

$$\hat{x} = \arg\min_x \left\{ \frac{1}{2\sigma^2} \|y - x\|^2 + \frac{1}{2} x^t B x \right\} \ . \tag{12}$$

We also note that this MAP cost function can be written in a number of different ways as shown below.

$$\begin{aligned} c(x) &= \frac{1}{2\sigma^2} \|y - x\|^2 + \frac{1}{2} x^t B x & (13) \\ &= \frac{1}{2\sigma^2} \|y - x\|^2 + \frac{1}{2\sigma_x^2} x^t (I - G) x & (14) \\ &= \frac{1}{2\sigma^2} \|y - x\|^2 + \frac{1}{2\sigma_x^2} \sum_{\{s,r\}} g_{s-r} (x_s - x_r)^2 \ , & (15) \end{aligned}$$

where again we use the assumptions that $B = \frac{1}{\sigma_x^2}(I - G)$, $G_{s,r} = g_{s-r}$, and $1 = \sum_r G_{s,r}$.

So, from this, we can see that the gradient of the cost function is given by

$$\begin{aligned} \nabla c(x) &= -\frac{1}{\sigma^2}(y - x) + Bx & (16) \\ &= -\frac{1}{\sigma^2}(y - x) + \frac{1}{\sigma_x^2}(I - G)x \ . & (17) \end{aligned}$$

## 3.1 Gradient Descent (GD) Optimization

The gradient decent (GD) algorithm works by sequentially updating the image, $x$, in the direction of the negative gradient of the cost function of (12). So the update is given by

$$x \leftarrow x + \alpha \left[ \frac{1}{\sigma^2}(y - x) - \frac{1}{\sigma_x^2}(I - G)x \right]$$

If we assume that the non-causal prediction coefficients have the following form,

$$G_{i,j} = \begin{cases} g_{i,j} & \text{for } j \in \partial i \\ 0 & \text{otherwise} \end{cases}$$

Then we can write this update in the following more explicit form.

$$x_s \leftarrow x_s + \alpha \left[ \frac{1}{\sigma^2}(y_s - x_s) - \frac{1}{\sigma_x^2} \left( x_s - \sum_{r \in \partial s} g_{s,r} x_r \right) \right]$$

Importantly, all these updates can be done in parallel in much the same way as an FIR filter is implemented.

The following provides a pseudo-code specification of the GD algorithm.

**Standard GD Algorithm:**

```
1. Select desired step size $\alpha$
2. Initialize $x \leftarrow y$
3. For $n = 0$ to $N_I - 1$
    (a) In parallel for each $s \in S$
            $x \leftarrow x + \alpha \left[ \frac{1}{\sigma^2}(y - x) - \frac{1}{\sigma_x^2}(x - Gx) \right]$
```

## Section Problems:

**Instructions:**
For the following problems, you are required to use parallelism techniques in JAX to meet the execution-time requirement. To do this, you should use the vmap function to vectorize the GD update.

**Problems:**

1. Show that the cost function of (13) is strictly convex.
2. Show that any local minimum of the cost function of (13) is also a global minimum.

3. Generate the noisy image $y = x + w$ where $x$ is the monochrome image obtained in the previous section and $w$ is i.i.d. Gaussian noise with mean zero and $\sigma = 0.05$. Then clip $y$ to the range $[0, 1]$ and print out the image $y$.
4. Compute the MAP estimate of $x$ using $N_I = 20$ iterations of GD optimization. Use $\sigma_x = 0.2$, $\sigma = 0.1$, and $\alpha = 0.001$.
   (a) Print out the resulting MAP estimate.
   (b) Plot the cost function of (13) as a function of the iteration number.
   (c) Report the total execution time of your script.
5. Repeat step 4 for $\sigma_x = 1.0$ and $\sigma_x = 0.04$.

## 3.2 Iterative Coordinate Descent (ICD) Optimization

The iterative coordinate decent (ICD) algorithm works by sequentially minimizing the cost function of (12) with respect to each pixel. In this case, the cost function for MAP estimation has the form

$$c(x) = \frac{1}{2\sigma^2}||y - x||^2 + \frac{1}{2\sigma_x^2}x^t(I - G)x \tag{18}$$

For a quadratic cost function, it can be shown that the ICD update can be written as

$$x_s \leftarrow x_s - \frac{\theta_{1,s}}{\theta_{2,s}} \ ,$$

where

$$\begin{aligned}\theta_{1,s} &= [\nabla c(x)]_s = \text{gradient at pixel } s \\ \theta_{2,s} &= [\nabla^2 c(x)]_{s,s} = \text{diagonal of Hessian at pixel } s\end{aligned}$$

The gradient is given in Equation 17, and the Hessian can be obtained by differentiating the gradient and is given below.

$$\begin{aligned}\theta_1 &= \nabla^2 c(x) \\ &= \nabla_x \left[\frac{1}{\sigma^2}(y - x) + \frac{1}{\sigma_x^2}(I - G)x\right] \\ &= \frac{1}{2\sigma^2}I + \frac{1}{2\sigma_x^2}(I - G)\end{aligned}$$

Using the fact that $\text{diag}[G] = 0$, we have that

$$\theta_2 = \text{diag}[\nabla^2 c(x)] = \left(\frac{1}{2\sigma^2} + \frac{1}{2\sigma_x^2}\right)$$

From this, we have

$$\theta_{1,s} = \left[ -\frac{1}{\sigma^2}(y-x) + \frac{1}{\sigma_x^2}(I-G)x \right]_s \tag{19}$$

$$= -\frac{1}{\sigma^2}(y_s - x_s) + \frac{1}{\sigma_x^2}\left( x_s - \sum_{r\in\partial s} g_{s,r}x_r \right) \tag{20}$$

$$\theta_{2,s} = \frac{1}{\sigma^2} + \frac{1}{\sigma_x^2} \tag{21}$$

This results in the ICD update for $x_s$ given by

$$x_s \leftarrow x_s + \frac{\frac{1}{\sigma^2}(y_s - x_s) - \frac{1}{\sigma_x^2}\left( x_s - \sum_{r\in\partial s} g_{s,r}x_r \right)}{\frac{1}{\sigma^2} + \frac{1}{\sigma_x^2}} .$$

Then the standard ICD algorithm is given by

**Standard ICD Algorithm:**

```
1. Initialize $x \leftarrow y$
2. For $n = 0$ to $N_I - 1$
   (a) Sequentially for each $s \in S$
       $x_s \leftarrow x_s + \frac{\frac{1}{\sigma^2}(y_s - x_s) - \frac{1}{\sigma_x^2}\left( x_s - \sum_{r\in\partial s} g_{s,r}x_r \right)}{\frac{1}{\sigma^2} + \frac{1}{\sigma_x^2}}$
```

## Section Problems:

### Instructions:

For the following problems, you are required to use JAX for loop in order to ensure sequential execution of the ICD pixel updates. For the MAP estimation portion, use the noisy image generated for problem 3 of the previous section.

For extra credit, use the *structured control flow primitives* in JAX along with JIT compiling to speed your code. See the following link for details:
`https://docs.jax.dev/en/latest/control-flow.html#structured-control-flow-primitives`

### Problems:

1. Show that the costs resulting from ICD updates form a monotone decreasing sequence that is bounded below.
2. Show that any local minimum of the cost function of (13) is also a global minimum.
3. Compute the MAP estimate of $X$ using $N_I = 20$ iterations of ICD optimization with $\sigma_x = 0.2$, $\sigma = 0.1$, and $\alpha = 0.001$.

(a) Print out the resulting MAP estimate.

(b) Plot the cost function of (13) as a function of the iteration number.

(c) Report the total execution time of your script.

## 3.3 Vectorized Coordinate Descent (VCD) Optimization

The limitation of ICD is that sequential operations are very slow on modern computers such as GPUs. In order to overcome this limitation, we can parallelize these updates using an algorithm which we call vectorized coordinate descent (VCD).

In order to do VCD updates, we first partition the set of pixels into $K$ subsets denoted by $S_k$ such that $S = \cup_{k=0}^{K-1} S_k$ is the set of all pixels, and $S_i \cap S_j =$ for $i \neq j$. Furthermore, we design each subset so that none of its members are neighbors. More specifically, we have that $\forall s, r \in S_k,\ r \notin \partial s$.

For example, if we are using a 4-point neighborhood on a 2D lattice, then we can select pixel subsets defined by

$$S_{(k_0,k_1)} = \{(s_0, s_1) \in S : (s_0 + k_0, s_1 + k_1)//3 = 0\} \ , \tag{22}$$

where $0 \leq k_0 < 3$ and $0 \leq k_1 < 3$. This results in $K = 3^2$ subsets denoted by $S_0, \ldots, S_8$ that partition the full set of pixels $S$. Importantly, any two pixels in the same subset are guaranteed not to be neighbors.

The VCD algorithm then updates all pixels in a partition, $S_k$, in parallel, but the partitions are processed sequentially. Since the pixels in a partition, $S_k$, share no neighbors, they can be updated independently and in parallel.

In order to simplify the representation of the algorithm, we also define the decimation operator $D_k$, for $k = 0, \ldots, 3$. So then $D_k : \Re^N \to \Re^{|S_k|}$ selects out the values in the subset $S_k$. Alternatively, $D_k^t$ takes the values in $S_k$ and inserts them into a vector of dimension $N$ with zeros in the unused locations. We also note that the functions $x \leftarrow D_k y$ and $x \leftarrow x + D_k^t y$ can be written in JAX respectively as

```
x = y[index_set]

x = x.at[index_set].add(y)
```

where `index_set` is a JAX array of indices for the set $S_k$.

Using this, we can then write the VCD update for the subset $S_k$ as

$$\begin{aligned}
\theta_1 &\leftarrow D_k \left[ -\frac{1}{\sigma^2}(y - x) + \frac{1}{\sigma_x^2}(I - G)x \right] \\
\theta_2 &\leftarrow D_k \mathbf{1} \left[ \frac{1}{\sigma^2} + \frac{1}{\sigma_x^2} \right] \\
x &\leftarrow x - D_k^t(\theta_1/\theta_2)
\end{aligned}$$

where $\mathbf{1}$ denotes a column vector of 1's and $(\cdot/\cdot)$ denotes element-wise division.

This results in the VCD algorithm shown in the following pseudo-code.

**Standard VCD Algorithm:**

```
1. Initialize $x \leftarrow y$
2. For $n = 0$ to $N_I - 1$:
    (a) Sequentially for $k = 0$ to $K - 1$:
        $\theta_1 \leftarrow D_k \left[ -\frac{1}{\sigma^2}(y - x) + \frac{1}{\sigma_x^2}(I - G)x \right]$
        $\theta_2 \leftarrow D_k \mathbf{1} \left[ \frac{1}{\sigma^2} + \frac{1}{\sigma_x^2} \right]$
        $x \leftarrow x - D_k^t(\theta_1/\theta_2)$
```

## Section Problems:

**Instructions:**

For the following problems, you are required to use parallelism techniques in JAX.

- Assume that $\partial s$ defines an 8-point neighborhood system in 2D.
- Use the vmap function to parallelize the updates of pixels in each subset.
- For the MAP estimation portion, use the noisy image generated for problem 3 of the previous section.

**Problems:**

1. Construct pixels subsets, $S_k$, from Equation 22.
2. Compute the MAP estimate of $X$ using 20 iterations of VCD optimization. Use $\sigma_x = 0.2$ and $\sigma = 0.05$.
   (a) Print out the resulting MAP estimate.
   (b) Plot the cost function of (13) as a function of the iteration number.
   (c) Report the total execution time of your script.

# 4 MAP Deconvolution with Gaussian Prior

In this section, you will restore an image that has been convolved with a linear filter and then corrupted by AWGN. For this problem, it is best to think of the images $Y$ and $X$ as being $N$ dimensional column vectors formed by listing the pixels in raster order. In this case, a linear space-invariant filter can be represented by a $N \times N$ matrix transform $A$.

## 4.1 The Deconvolution Forward Model

So the forward model of $Y$ given $X$ has the form

$$Y = A\,X + W$$

where $A$ is a Toeplitz-block-Toeplitz matrix[1] that implements a 2-D shift invariant filter applied with clamp boundary conditions, and $W \sim N(0, \sigma_x^2 I)$ is AWGN.

For this problem, we will assume that the linear filter $A$ has the form of FIR convolution with a filter $h(n,m)$ that is of size $(2P+1) \times (2P+1)$ where $P = 5$ and

$$h(n,m) = \frac{1}{2(2P+1)} \left[ \sum_{k=-P}^{P} \delta(n-k)\delta(m) + \sum_{l=-P}^{P} \delta(m-l)\delta(n) \right] .$$

Intuitively, for $P = 5$ the inpulse response $h(n,m)$ has the form

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1/22 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1/22 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1/22 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1/22 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1/22 | 0 | 0 | 0 | 0 | 0 |
| 1/22 | 1/22 | 1/22 | 1/22 | 1/22 | 2/22 | 1/22 | 1/22 | 1/22 | 1/22 | 1/22 |
| 0 | 0 | 0 | 0 | 0 | 1/22 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1/22 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1/22 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1/22 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1/22 | 0 | 0 | 0 | 0 | 0 |

(23)

where the 2/22 term is at the center of the filter.

In this case, the cost functional for MAP estimation has the form

$$c(x) = \frac{1}{2\sigma^2}||y - Ax||^2 + \frac{1}{2\sigma_x^2} x^t (I - G) x \tag{24}$$

The VCD applies ICD updates for every pixel in the subset $S_k$. However, since every pixel update is a greedy update of the cost function, the parallel update of many pixels may be too large. In practice, this can mean that the cost function increases after a parallel ICD update, rather than decreasing, causing the algorithm to become unstable. So in order to ensure convergence of the VCD algorithm, we will include a damping factor, $\alpha$, that optimally scales the size of the parallel ICD updates.

[1] You can use either circular or clamped boundary conditions for implementation of $A$, whichever is easier. But whatever choice you make, you need to be consistent.

Referring back to Section 3.3, the gradient and diagonal Hessian are given by

$$\begin{aligned}\theta_1 &\leftarrow \nabla c(x) = -\frac{1}{\sigma^2}A^t(y-Ax) + \frac{1}{\sigma_x^2}(I-G)x \\ \theta_2 &\leftarrow \text{Diag}\left[\nabla^2 c(x)\right] = \left[\frac{1}{\sigma^2}\|A_{*,s}\|^2 + \frac{1}{\sigma_x^2}\right]_{s=0}^{N-1} .\end{aligned}$$

Notice, that for circular boundary conditions, $\theta_{2,s} = \textit{constant}$, however for the truncated boundary conditions, the value of $\|A_{*,s}\|^2$ will depend on $s$.

## 4.2 The VCD Update Equations

We next derive the VCD update equations for this optimization problem. The the ICD update for a single pixel is given by

$$x_s \leftarrow x_s - \frac{\theta_{1,s}}{\theta_{2,s}} .$$

To reduce computation, we will need to keep a state vector, $e = y - Ax$, which we will update after each iteration. Using this framework, the VCD update is given by the parallel application of ICD updates at the subset of pixels being updated. So, the VCD update for the pixel subset $S_k$ is given by

$$\begin{aligned}\theta_1 &\leftarrow \left[-\frac{1}{\sigma^2}A_k^t e + \frac{1}{\sigma_x^2}D_k(I-G)x\right] \\ d &\leftarrow -\frac{\theta_1}{D_k\theta_2} \\ p &\leftarrow A_k d \\ d &\leftarrow D_k^t d \\ x &\leftarrow x + \alpha d \\ e &\leftarrow e - \alpha p\end{aligned}$$

where $\alpha$ is a possible damping factor we may need to use to stabilize the algorithm $A_k$ and $A_k^t$ are sparse forward and back projection operators defined by

$$A_k = AD_k^t \tag{25}$$

$$A_k^t = D_k A^t . \tag{26}$$

We will give more details on how to efficiently implement these linear operators in Section 4.3 below.

We can compute the optimal value of the dampling parameter, $\alpha^*$, as

$$\alpha^* = \arg\min_\alpha c(x+\alpha d) = \frac{-b}{a} ,$$

where

$$b = -\frac{1}{\sigma^2} e^t p + \frac{1}{\sigma_x^2} x^t (I - G) d \tag{27}$$

$$a = \frac{1}{\sigma^2} \|p\|^2 + \frac{1}{\sigma_x^2} d^t (I - G) d \tag{28}$$

This results in the following VCD algorithm for deblurring with additive noise and a Gaussian prior.

```
VCD Algorithm with Blurring Filter:
    1. Initialize $x \leftarrow y$
    2. Initialize $e \leftarrow y - Ax$
    3. Initialize $\theta_2$ with
           $\theta_{2,s} \leftarrow \frac{1}{\sigma^2}\|A_{*,s}\|^2 + \frac{1}{\sigma_x^2}$
    4. For $n = 0$ to $N_I - 1$
        (a) Sequentially for $k = 0$ to $K - 1$:
               $\theta_1 \leftarrow \left[ -\frac{1}{\sigma^2} A_k^t e + \frac{1}{\sigma_x^2} D_k (I - G) x \right]$
               $d \leftarrow -(\theta_1/\theta_2)$
               $p \leftarrow A_k d$
               $d \leftarrow D_k^t d$
               $b \leftarrow -\frac{1}{\sigma^2} e^t p + \frac{1}{\sigma_x^2} x^t (I - G) d$
               $a \leftarrow \frac{1}{\sigma^2} \|p\|^2 + \frac{1}{\sigma_x^2} d^t (I - G) d$
               $\alpha \leftarrow -b/a$
               $x \leftarrow x + \alpha d$
               $e \leftarrow e - \alpha p$
```

## 4.3 Computing Sparse Projectors on Parallel Architectures

Modern computers, such as GPUs, typically have many thousands of processors (i.e., cores) that operate in parallel. So, it is important that algorithms be designed to keep all these processors busy. A canonical operation for parallelization is the matrix-vector product

$$y = Ax \ ,$$

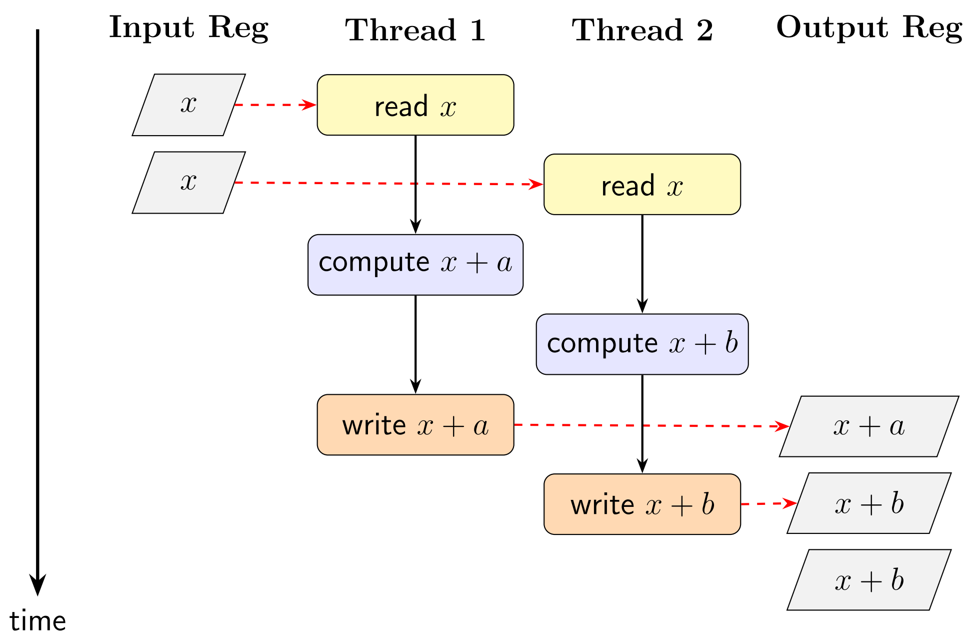

Figure 1: Illustration of race condition cause by parallel thread implementation of $x + a + b$. Both threads read the same value $x$. Then Thread 2 writes overwrites the final value of $x + b$. Notice that Thread 1's update of $x + a$ is lost.

where $A$ is an $m \times n$ matrix.

Conceptually, the matrix-vector product can be computed by iterating over columns or rows of the matrix $A$ as shown in Algorithms 1 and 2, respectively. However, in practice, iterating over rows is much better than iterating over columns because the row iterations can be done in parallel and the column iterations cannot. The reason that column iterations cannot be done in parallel is because this requires that multiple threads add separate values to the same output entry of $y_j$, which causes a race condition.

**Algorithm 1** Matrix-vector product $y = Ax$ (column-wise, not parallelizable)

```
1: $y \leftarrow 0$
2: for $j = 1$ to $n$ do                         ▷ Loop over columns of $A$
3:     $y \leftarrow y + A_{*,j}\, x_j$
4: end for
```

**Algorithm 2** Matrix-vector product $y = Ax$ (row-wise, parallelizable)

```
1: for $i = 1$ to $m$ do in parallel             ▷ Loop over rows of $A$
2:     $y_i \leftarrow A_{i,*}\, x$
3: end for
```

Figure 4.3 illustrates the problem that can occur when accumulating $x + a + b$ using two parallel threads running asynchronously on a computer. Notice that if Thread 2 reads its input value before Thread 1 saves the value $x + a$, then Thread 2 writes over the output with

```python
# Define back projection of pixel
def back_project_pixel(i, A, y):
    return jnp.dot(A[:, i], y)

# Vectorize over i, the pixel index
sparse_AT_fn = jax.jit(jax.vmap(back_project_pixel, in_axes=(0,
   None, None)))

# Construct sparse forward projection with jax.linear_transpose
@jax.jit
def sparse_A_fn(idx_, A_, x_sparse_):
    def f_y(y__):
        return sparse_AT_fn(idx_, A_, y__)
    y0 = jnp.zeros((A_.shape[0],), dtype=A_.dtype)
    adjoint_fn = jax.linear_transpose(f_y, y0)
    (z_,) = adjoint_fn(x_sparse_)
    return z_

# Compute sparse back and forward projections
x_sparse = sparse_AT_fn(idx, A, y)
z = sparse_A_fn(idx, A, x_sparse)
```

Figure 2: JAX code for efficiently computing sparse back and forward projectors.

the incorrect value $x + b$. So from this, we know that in order to avoid race conditions, we will need to apply the JAX vmap function to parallelize the output rather than the input of the variables of a linear operator.

For VCD, we will need to process sparse subsets of pixels which can be represented by the following two operators.

$$A_k = AD_k^t \tag{29}$$
$$A_k^t = D_k A^t \ , \tag{30}$$

where $A_k$ applies the linear operator to a sparse subset of pixels, and $A_k^t$ is its adjoint operator. More specifically, we have that

$$[A_k x]_i = \sum_{j \in S_k} A_{i,j} x_j \tag{31}$$
$$[A_k^t y]_i = \sum_{i} A_{i,j} y_i \text{ for } i \in S_k \ . \tag{32}$$

So in order to parallelize these operations, we will need to vectorize over the sparse set of pixels $i \in S_k$. But since this vectorization is best done in the *output* variable of the transformation, this implies that it is better to vectorize the back projection operation of $A_k^t$ first. We can then compute $A_k$ by taking the adjoint of $A_k^t$. Perhaps this is a bit surprising, but effective.

Figure 2 provides a more concrete example of this approach using a portion of a JAX script. The full JAX Python script for this example is provided on the lab web page. Notice that the script first constructs the back projection operator $A_k^t$ by applying `jax.vmap` over the sparse set of pixels specified by the list `idx`. It then constructs the forward projection operator using the JAX function `jax.linear_transpose`. Importantly, all modern machine learning languages, such as JAX, PyTorch, and Julia, support automatic differentiation in order to do optimization of neural networks. These optimization proceedures require the automatic computation of the adjoint gradient used in back propagation. The `jax.linear_adjoint` function is further optimized to take advantage of the fact that the underlying function is linear.

Line 6 of the code constructs a sparse back-projector by vectorizing over the pixel index. This vectorization is specified with the syntax `in_axes=(0, None, None)` where the "0" indicates that the first argument is "batched". Then when `sparse_AT_function`

Lines 9 to 16 construct the sparse forward-projector by taking the adjoint of the back projector using the `jax.linear_transpose` operations. This ensures that `sparse_A_fn` is an exact adjoint of `sparse_AT_fn` while also being efficient to compute.

## Section Problems:

**Instructions:**

For the following problems, you are required to use parallelism techniques in JAX to meet the execution-time requirement. To do this, you should use the vmap function to vectorize the ICD update.

In particular, you solution should meet the following requirements:

1. The script's execution time must be less than 30 seconds.
2. The plotted cost function in Equation 24 must decrease monotonically as the number of iterations increases.

**Problems:**

1. Construct pixels subsets, $S_k$, from Equation 22.
2. Compute the MAP estimate of $X$ for this deblurring problem using 20 iterations of VCD optimization. Use $\sigma_x = 0.2$ and $\sigma = 0.05$.
   (a) Print out the resulting MAP estimate.
   (b) Plot the cost function of (13) as a function of the iteration number.
   (c) Report the total execution time of your script.