A note on asynchronous Projective Splitting in Julia

Utkarsh Sharma[a], Kashish Goel[a], Aryan Dua[a], Sebastian Pokutta[b], Zev Woodstock[b,c]

[a]*Department of Computer Science and Engineering, I.I.T. Delhi, Hauz Khas, New Delhi, 110016, India*
[b]*Department of AI in Society, Science, and Technology, Zuse Institute Berlin, Takustr. 7, 12045, Berlin, Germany*
[c]*Department of Mathematics and Statistics, James Madison University, MSC 1911, 60 Bluestone Dr., Harrisonburg, Virginia, 22807, USA*

Abstract

While it has been mathematically proven that Projective Splitting (PS) algorithms can converge in parallel and distributed computing settings, to-date, it appears there were no open-source implementations of the full algorithm with asynchronous computing capabilities. This note fills this gap by providing a Julia implementation of the asynchronous PS algorithm of Eckstein and Combettes for solving fully nonsmooth convex optimization problems. Our methodology includes inter-operability with existing packages within the Julia ecosystem, and we also document observations from running this algorithm asynchronously on problems in image processing and machine learning.

Keywords: nonsmooth, convex, optimization, asynchrony, parallel computing

1. Introduction

The rapid advancement of data-driven applications have intensified the need for efficient algorithms capable of addressing first-order nonsmooth optimization problems, e.g., those appearing in machine learning, signal processing, and operations research [1, 2, 3, 4]. Despite the critical importance of first-order nonsmooth optimization, the landscape of available algorithms remains somewhat limited, particularly when considering algorithms which are both (A) proven to converge and (B) designed to leverage parallel and distributed computing environments. In this article, we consider a promising

recent class of algorithms which satisfies both of these criterion – *projective splitting* (PS) algorithms.

In 2008, Eckstein and Svaiter proposed the *projective splitting* framework for finding a zero of a sum of maximally monotone operators, and in particular, for minimizing a sum of nonsmooth convex functions [5]. In recent years, projective splitting algorithms have received attention from the optimization community, with many variants and new convergence rates being established [6, 7, 8]. Perhaps one of the first major theoretical breakthroughs for this class of algorithm came in 2016–2018, when it was proven that allowing for parallelized updates with bounded asynchrony is still guaranteed to converge to a solution of the mathematical optimization problem at-hand [9, 10]. In some sense, this capability sets projective splitting algorithms apart from other convex optimization algorithms: Even if we only consider the synchronous version of PS, there are several desirable properties exhibited by PS (e.g., block-iterativeness, fully nonsmooth capabilities, and no requirement of linear operator bounds)which, it seems, do not simultaneously occur for other algorithms [11]. Hence, in combination with the benefits outlined in [11] (whose scope only involved synchronous algorithms), the capability for asynchrony sets projective splitting apart from other algorithms. Its full potential was used to develop a specialized algorithm for the task of progressive hedging in [12]. However, aside from the specialized application in [12], it appears that all of the computational experiments involving projective splitting algorithms thus-far have focused on the *synchronous* case [11, 8]. Furthermore, there is a lack of literature discussing general implementation of the algorithms theoretically proven to converge in [9, 10]. For instance, PS is proven to converge for a wide range of hyperparamers; however, for some applications, the hyperparameters can drastically influence performance. Our main goal of this article is to (A) fill this gap in the literature by sharing our experience in using asynchronous algorithms on applications in classifier training and image processing, and (B) provide an open-source software implementation of the asynchronous parallel PS algorithm of [10].

1.1. Literature Review

One computational study was performed by [11], where it was established that the synchronous algorithms [13] and [10] (in synchronous mode) appear to be the only two methods with certain desirable properties for mathematical optimization (e.g., the abilities to fully split the mathematical optimization

problem, handle nonsmoothness, and omit estimates for the norm of the linear operators). It was shown that, in several applications in machine learning and image processing, [10] appears to out-perform [13] in terms of wall-clock time. However, the experiments in [11] were entirely for the synchronous version of the projective splitting algorithm in [10]. A more recent variant of projective splitting is also proven to allow for parallel block-asynchronous updates [8]. While the article includes very impressive experiments, it again does not study the impact of asynchrony.

It appears that the only current implementation of projective splitting with asynchrony implemented is for the specific application to the progressive hedging method of Rockafellar and Wets [12]. The method studied in [12], which is a special case of [9, 10], is shown to perform exceedingly well for this particular application. This work is encouraging for us to develop a general software implementation of [10] *with asynchrony* for the use in all applications, not just that of progressive hedging.

2. Background

We begin with preliminaries; for further background, see [14].

2.1. Mathematical Optimisation

Let $\mathcal{H}$ be a real, finite dimensional Hilbert Space with norm $\|\cdot\|$ and inner product $\langle\cdot \mid \cdot\rangle$. We define the extended real line $[-\infty, +\infty]$ as $(-\infty, \infty) \cup \{-\infty, +\infty\}$. For algebra on the extended real line, for the purposes of optimisation, we are mainly concerned with defining addition, a binary operator such that for $x \in \mathbb{R}, x + \infty = \infty$ and $\infty + \infty = \infty$.

For $f : \mathcal{H} \to [-\infty, +\infty]$, we are interested in finding

$$\operatorname*{argmin}_{x \in \mathcal{H}} f(x)$$

Definition 2.1. *The proximity operator* $\operatorname{Prox}_{\gamma f}(x)$ *of a function* $f \colon \mathcal{H} \to [-\infty, +\infty]$ *at point* x *with parameter* $\gamma > 0$ *is defined as:*

$$\operatorname{Prox}_{\gamma f}(x) = \operatorname*{argmin}_{y \in \mathcal{H}} \left\{ f(y) + \frac{1}{2\gamma} \|y - x\|_2^2 \right\}.$$

The simplest example of using proximity operators for minimisation is perhaps [15] showing the sequence formed by the recursive application of $\operatorname{Prox}_{\gamma f}$

converges to a minimizer of f. As is well-known in the optimization community, computing $\text{Prox}_{\Sigma f_i}$, i.e., the proximity operator for the sum Σf_i, is oftentimes computationally expensive or intractable; on the other hand, evaluating the individual operators $(\text{Prox}_{f_i})_{i\in I}$ is far easier. Minimising sums via the use of the individual proximity operators is called *splitting*.

We implement Algorithm 4 of [10] which, in addition to being a splitting algorithm, is also block-activated, and asynchronous. For minimising the sum of functions Σf_i for $i \in \{1, ..., m\}$, computing $\text{Prox}_{\gamma f_i}$ for each i, may still be prohibitively slow. *Block-activated* (or *block-iterative*) algorithms activate a subset $I_n \subset \{1, ..., m\}$ of the proximity operators, during the n^{th} iteration. Asynchrony in our algorithm allows us to compute proximity operators $\text{Prox}_{\gamma f_i}$ for $i \in I_{n+1}$ without waiting for the proximity operators to be computed in I_n.

2.2. Notation and problem formulation

Our notation and definitions are standard in continuous optimization; for further background see, e.g., [14]. We will use $\mathcal{H}$ to represent a Hilbert Space with inner product $\langle \cdot \mid \cdot \rangle$ and $\| \cdot \| = \sqrt{\langle \cdot \mid \cdot \rangle}$. For most problems our $\mathcal{H}$ will be $\mathbb{R}^n$ with inner product as the usual Euclidean dot product. The direct sum is defined as $\bigoplus_{i=1}^m \mathcal{H}_i = \mathcal{H}_1 \times \ldots \times \mathcal{H}_m$, where $\mathcal{H}_i$ represents the i^{th} Hilbert space. The inner product on the direct sum is defined as $\langle (x_i)_{i=1}^m, (y_i)_{i=1}^m \rangle = \sum_{i=1}^m \langle x_i, y_i \rangle_{\mathcal{H}_i}$ where $(x_i)_{i=1}^m, (y_i)_{i=1}^m \in \bigoplus_{i=1}^m \mathcal{H}_i$. The set of functions from $\mathcal{H}$ to $[-\infty, +\infty]$ that are convex, lower-semicontinuous, and proper is denoted $\Gamma_0(\mathcal{H})$.

Our implementation of [10] minimises an objective function of the following form

$$\underset{(x_i)_{i\in I} \in \bigoplus \mathcal{H}_i}{\text{minimize}} \quad \sum_{i\in I} f_i(x_i) + \sum_{k\in K} g_k\Big(\sum_{i\in I}(L_{ki} \cdot x_i)\Big) \tag{1}$$

where $f_i : \mathcal{H}_i \to \mathbb{R}$ with $f_i \in \Gamma_0(\mathcal{H}_i)$ and $g_k : \mathcal{G}_k \to \mathbb{R}$ with $g_k \in \Gamma_0(\mathcal{G}_k)$. $(\mathcal{H}_i)_{i\in I}$ and $(\mathcal{G}_k)_{k\in K}$ are real Hilbert spaces with $I = \{1, .., m\}$, $K = \{1, .., p\}$ and $L_{ki} : \mathcal{H}_i \to \mathcal{G}_k$ are linear operators $\forall k \in K, i \in I$. From here on, we will use $g_k(x)$ to represent the splitting functions that take in a linear combination of transformed x_i as their inputs.

2.3. The variational Combettes-Eckstein projective splitting algorithm

The variational Combettes-Eckstein projective splitting algorithm is proven to converge under the following assumptions [10].

Assumption 2.2. *For every $i \in I$ and every $k \in K$, let $(c_i(n))_{n\in\mathbb{N}}$ and $(d_k(n))_{n\in\mathbb{N}}$ be the sequences in $\mathbb{N}$ that represent the most recent iterations for which (A) computations $\operatorname{Prox}_{f_i}$ or $\operatorname{Prox}_{g_k}$ were respectively launched, and (B) their computation has completed by the current iteration n.*

(i) *A solution to (1) exists.*

(ii) *For every $i \in I$ and $k \in K$, we have $f_i \in \Gamma_0(\mathcal{H})$ and $g_k \in \Gamma_0(\mathcal{G})$.*[1]

(iii) *There exists a strictly positive integer M such that*

$$\forall n \in \mathbb{N}, \bigcup_{j=n}^{n+M-1} I_j = I \text{ and } \bigcup_{j=n}^{n+M-1} K_j = K. \tag{2}$$

(iv) *There exists a positive integer D such that for every iteration $n \in \mathbb{N}$, and all indices $i \in I$, $k \in K$, we have*

$$n - D \leq c_i(n) \leq n \text{ and } n - D \leq d_k(n) \leq n. \tag{3}$$

(v) *The hyperparameters $\gamma_{i,n}$ and $\mu_{k,n}$ are bounded away from 0 and ∞. That is,*

$$0 < \liminf_{n\to\infty} \gamma_{i,n} \leq \limsup_{n\to\infty} \gamma_{i,n} < \infty, \tag{4}$$

$$0 < \liminf_{n\to\infty} \mu_{k,n} \leq \limsup_{n\to\infty} \mu_{k,n} < \infty. \tag{5}$$

Here, (iii) ensures that for some positive integer M, every M consecutive blocks cover the entire set. Further, (iv) ensures that at any current iteration, no prox computation that is left is older than D.

[1] usually $\mathcal{H}$ and $\mathcal{G}$ are of the form $\bigoplus_{i=1}^{m} \mathbb{R}^{n_i}$

Algorithm 1 Combettes-Eckstein Algorithm [10]

Require: $I_0 = \{1, \ldots, m\}$ and $K_0 = \{1, \ldots, p\}$. Suppose Assumption 2.2 holds and $(c_i(n))_{n\in\mathbb{N}}$ and $(d_k(n))_{n\in\mathbb{N}}$ are sequences in $\mathbb{N}$ as defined in 2.2. For every $i \in \{1, \ldots, m\}$ and every $k \in \{1, \ldots, p\}$, let $\{\gamma_{i,n}, \mu_{k,n}\} \subset]0, +\infty[$, $x_{i,0} \in \mathcal{H}_i$, and $v^*_{k,0} \in \mathcal{G}_k$.

for $n = 0, 1$ **to** $\ldots$ **do**
  $\lambda_n \in]0, 2[$
  if $n > 0$ **then**
    Select $\varnothing \neq I_n \subset \{1, \ldots, m\}$ and $\varnothing \neq K_n \subset \{1, \ldots, p\}$
  end if
  for $i \in I_n$ **do**
    $x^*_{i,c_i(n)} = x_{i,c_i(n)} - \gamma_{i,c_i(n)} \sum_{k=1}^{p} L^*_{k,i} v^*_{k,c_i(n)}$
    $a_{i,n} = \operatorname{Prox}_{\gamma_{i,c_i(n)} f_i} x^*_{i,c_i(n)}$
    $a^*_{i,n} = \gamma^{-1}_{i,c_i(n)}(x^*_{i,c_i(n)} - a_{i,n})$
  end for
  $(a_{i,n}, a^*_{i,n})_{i\in\{1,\ldots,m\}\smallsetminus I_n} = (a_{i,n-1}, a^*_{i,n-1})_{i\in\{1,\ldots,m\}\smallsetminus I_n}$
  for $k \in K_n$ **do**
    $y^*_{k,n} = \mu_{k,d_k(n)} v^*_{k,d_k(n)} + \sum_{i=1}^{m} L_{k,i} x_{i,d_k(n)}$
    $b_{k,n} = \operatorname{Prox}_{\mu_{k,d_k(n)} g_k} y^*_{k,n}$
    $b^*_{k,n} = \mu^{-1}_{k,d_k(n)}(y^*_{k,n} - b_{k,n})$
  end for
  $(b_{k,n}, b^*_{k,n})_{k\in\{1,\ldots,p\}\smallsetminus K_n} = (b_{k,n-1}, b^*_{k,n-1})_{k\in\{1,\ldots,p\}\smallsetminus K_n}$
  $(t_{k,n})_{k\in\{1,\ldots,p\}} = (b_{k,n} - \sum_{i=1}^{m} L_{k,i} a_{i,n})_{k\in\{1,\ldots,p\}}$
  $(t^*_{i,n})_{i\in\{1,\ldots,m\}} = (a^*_{i,n} + \sum_{k=1}^{p} L^*_{k,i} b^*_{k,n})_{i\in\{1,\ldots,m\}}$
  $\tau_n = \sum_{i=1}^{m} \|t^*_{i,n}\|^2 + \sum_{k=1}^{p} \|t_{k,n}\|^2$
  if $\tau_n > 0$ **then**
    $\pi_n = \sum_{i=1}^{m} \big(\langle x_{i,n} \mid t^*_{i,n}\rangle - \langle a_{i,n} \mid a^*_{i,n}\rangle\big) + \sum_{k=1}^{p} \big(\langle t_{k,n} \mid v^*_{k,n}\rangle - \langle b_{k,n} \mid b^*_{k,n}\rangle\big)$
  end if
  if $\tau_n > 0$ and $\pi_n > 0$ **then**
    $\theta_n = \lambda_n \pi_n / \tau_n$
    $(x_{i,n+1})_{i\in\{1,\ldots,m\}} = (x_{i,n} - \theta_n t^*_{i,n})_{i\in\{1,\ldots,m\}}$
    $(v^*_{k,n+1})_{k\in\{1,\ldots,p\}} = (v^*_{k,n} - \theta_n t_{k,n})_{k\in\{1,\ldots,p\}}$
  else
    $(x_{i,n+1})_{i\in\{1,\ldots,m\}} = (x_{i,n})_{i\in\{1,\ldots,m\}}$
    $(v^*_{k,n+1})_{k\in\{1,\ldots,p\}} = (v^*_{k,n})_{k\in\{1,\ldots,p\}}$.
  end if
end for

Theorem 2.3 ([10, Theorem 5]). *Consider the problem of* (1) *using Algorithm 1 under Assumptions 2.2. Then the sequences* $\mathbf{x}_n = (x_{i,n})_{i\in I}$ *and* $\mathbf{a}_n = (a_{i,n})_{i\in I}$ *converge to a solution of* (1).

3. Methodologies

We engineered our application to be compatible with `Julia 1.9.0` and higher versions. It is compatible with any proximal operators of the type defined in `ProximalOperators.jl`[2]. We also allow for usage of custom prox operators for complex functions in the format prescribed by the library wherever needed. To handle asynchronous programming, we make use of Julia's `Distributed`[3] library, spawning P processors and cyclically assigning prox computations over them.

The code requires the following inputs

(i) m, p : to describe the blocks $I = [m], K = [p]$.

(ii) $(f_i)_{1\leq i\leq m}, (g_k)_{1\leq k\leq p}$: the definitions for functions corresponding to (1). The prox computation for non-standard functions can also be added in the format prescribed by `ProximalOperators.jl`.

(iii) L : Abstract matrix consisting of operators L_{ki} used inside (1). The operators L_{ki} can be input either as matrices or as functions. We add support of `LinearAlgebra.jl`[4] for the use of these operators.

(iv) L^* : Abstract matrix consisting of the adjoint operators L^*_{ki}.

(v) D : the maximum delay allowed in iterations. As defined in Assumption 2.2.

(vi) I_n, K_n : in the form of functions that return the corresponding blocks for the n^{th} iteration.

(vii) $\gamma_{i,n}, \mu_{i,n}$: In the form of functions `generate_gamma(i,n)` and `generate_mu(i,n)`.

We implement the algorithm to be able to handle inputs with variable sizes i.e., for $x \in \bigoplus_{i=1}^m \mathcal{H}_i$, the input can belong to $\bigoplus_{i=1}^m \mathbb{R}^{k_i}$ where k_i can be all different. The L matrix consisting of L_{ki} operators mentioned in the original equation, can be inputted as a matrix of operators of the form of

[2] `https://juliafirstorder.github.io/ProximalOperators.jl/latest/`

[3] `https://docs.julialang.org/en/v1/stdlib/Distributed/`

[4] `https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/`

both - matrices and functions. We allow this flexibility so that when L_{ki} operator is a matrix, the adjoints can be simply handled as the transpose; on the other hand, when inputting operators as a function which computes matrix vector products, a means to compute of their adjoint operators (L_{ki}^*) must be provided.

The blocks (I_n), where n is the iteration count, can be described by the user as a function of n. We provide standard block functions $(I_n)_{n\in\mathbb{N}}$ (for $|I| = m$) such as:

(i) Full activation: $I_n = I$.
(ii) Cyclic activation: $I_n = \{n \mod m\}$.
(iii) Cyclic $\frac{1}{M}$ activation : $I_n = \{k, k+1, ..., k + \lfloor \frac{(n-1)m}{M} \rfloor - 1\}$ with appropriate k to ensure $\cup_{n=1}^{M} I_n = I, I_{n+M} = I_n$.

We also provide mechanisms to record observations such as $||x_n - x_{n-1}||$, $||x_n - x_\infty||$, function values and more over - epochs (refer Remark 3.1 for definition), iterations or prox-calls. These quantities are used to estimate optimality of the current iterate, speed of convergence and compare relative optimality respectively over different variables. We used $x_{2*\texttt{total\_iters}}$ for approximating x_∞ by default (but this value can easily be modified by the user).

Remark 3.1. *We call one* epoch *as the set of iterations during which every prox computation for each function in $(f_i)_{i\in I}$ and $(g_k)_{k\in K}$ has been performed at least once.*

4. Experiment Setup and Results

Two problem settings (Sections 4.1–4.2) were used in our experiments (Sections 4.3–4.4). Computations were performed on a Dell PowerEdge R470 system, running Linux with HPC jobs managed by slurm, equipped with 1500 GB of RAM, 24 cores, and 48 threads. The system runs on Intel(R) Xeon(R) Gold 6246 processors. The implementation and experimentation code can be found in our repository `https://github.com/zevwoodstock/AsyncProx/`.

4.1. Problem 1: Sparse linear classifier training

We consider the classification problem of [16], known as the *latent group lasso*, or *lasso with group overlap*. Let $\{G_1, \ldots, G_m\}$ be a covering of $\{1, 2, \ldots, d\}$.

Define $X = \{x_1, \ldots, x_m | x_i \in \mathbb{R}^d, \text{support}(x_i) \subset G_i\}$. The ideal classification vector is $\tilde{y} = \Sigma_{i=1}^m x_i$ where $(x_i)_{i=1}^m$ is a solution of

$$\underset{\mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_m \in X}{\arg\min} \sum_{i=1}^{m} \|\mathbf{x}_i\|^2 + \sum_{k=1}^{p} g_k \left(\sum_{i=1}^{m} \langle \mathbf{x}_i, \boldsymbol{\mu}_k \rangle \right), \tag{6}$$

where $\mu_k \in \mathbb{R}^d$, $g_k(\xi) = 10 \max\{0, 1 - \beta_k \xi\}$, where $\beta_k = \omega_k \text{sign}(\langle y \mid \mu_k \rangle)$ is the k^{th} measurement of the true vector $y \in \mathbb{R}^d$ ($d = 10,000$). To generate μ_k, we sampled $\mathbf{v} \sim \mathcal{N}(0, I_d)$, where I_d is the identity matrix, and set $\mu_k = \mathbf{v}/\|\mathbf{v}\|_2$. The values $\omega_k \in \{-1, 1\}$ are selected so that 25% of them (selected uniformly at random) are misclassified. The number of measurements made on true value y that we want to approximate is $p = 1000$. There are $m = 1429$ groups. For every $i \in \{1, ..., m-1\}$ each G_i has 10 consecutive integers and an overlap with G_{i+1} of 3.

We observe that the support vectors x_i are sparse with only $|G_i| = 10$ non-zero elements out of d total. To improve the memory requirements for this problem, $\mathbf{x}_i$ is replaced by $\tilde{\mathbf{x}}_i$ in (6) where $\tilde{\mathbf{x}}_i \in \mathbb{R}^{10}$ and $\tilde{X}$ is the set containing the compressed support vectors $\tilde{x}_i$. Thus we can use $F : \mathbb{R}^{10} \to \mathbb{R}^d$ such that $\forall i \in \{1, ..., m-1\}$, $F(\tilde{\mathbf{x}}_i) = \mathbf{x}_i$, i.e., it pads $\tilde{\mathbf{x}}_i$ with zeros.consisting

To construct this and an instance of Equation (1), we rewrite such that $(\tilde{x}_i)_{i=1}^m$ is given by

$$\underset{\tilde{\mathbf{x}}_1,\ldots,\tilde{\mathbf{x}}_m \in X}{\arg\min} \sum_{i=1}^{m} \|\tilde{\mathbf{x}}_i\|^2 + \sum_{k=1}^{p} g_k \left(\langle y_k, \boldsymbol{\mu}_k \rangle \right), \tag{7}$$

where $\tilde{x}_i \in \mathbb{R}^{10}, \mu_k \in \mathbb{R}^d$ and $y_k = \sum_{i \in I} (L_{ki} \cdot \tilde{\mathbf{x}}_i)$, $L_{ki} : \mathbb{R}^{10} \to \mathbb{R}^d$ such that $L_{ki}\tilde{\mathbf{x}}_i = F(\tilde{\mathbf{x}}_i)$. This can then be formulated into (1) by setting $f_i : \mathcal{H}_i = \mathbb{R}^{10} \to (-\infty, \infty]$ and $g_k : \mathcal{G}_k = \mathbb{R}^d \to (-\infty, \infty]$ such that, for every $1 \leq i \leq m$ and every $1 \leq k \leq p$, $f_i : x_i \mapsto \|x_i\|^2$ and $g_k : y_k \mapsto 10 \max\{0, 1 - \beta_k \langle y_k, \mu_k \rangle\}$, where $y_k = \sum_{i \in I} (L_{ki} \cdot x_i)$ and $L_{ki} = F$. We can check that the adjoint of our L_{ki} operator is $L_{ki}^* : \mathbb{R}^d \to \mathbb{R}^{10}$ such that for $y \in \mathbb{R}^d, L_{ki}^*(y) = (y_{7i+1}, y_{7i+2}, ..., y_{7i+10})$.

The operators Prox_{f_i} are available in `ProximalOperators.jl`, and Prox_{g_k} are calculated using [14, Proposition 24.14].

4.2. Problem 2: Image Recovery

We consider a Stereoscopic Image Recovery problem akin to [17, Section 4.2] where, instead of restoring a pair of images, we restore a series of

M images $\{x_i\}_{1\leq i\leq M}, x_i \in \mathbb{R}^N$. Noisy degraded versions are available via

$$z_i = \mathcal{L}x_i + w_i, \quad w_i \sim \mathcal{N}(0, \sigma^2\mathbf{I}),$$

where $\mathcal{L}$ blurs via convolution with a 5×5 averaging kernel with equal weights, and w_i is additive Gaussian noise with mean zero and variance $\sigma^2 = 0.0001$. The stereoscopy $x_i \approx D_i x_{i+1}$ is modeled by successive horizontal shift operators $D_i : \mathbb{R}^N \mapsto \mathbb{R}^N \ \forall i \in [M-1]$ with estimated shift values. We seek to solve

$$\underset{x_i\in\mathbb{R}^N, 1\leq i\leq M}{\arg\min} \sum_{i=1}^{M}\sum_{k=1}^{N} \phi_{i,k}(\langle x_i|e_{i,k}\rangle) + \sum_{i=1}^{M} \frac{1}{2\sigma^2}||\mathcal{L}x_i - z_i||^2 + \sum_{i=1}^{M-1} \frac{v}{2}||x_i - D_i x_{i+1}||^2, \tag{8}$$

where $(e_{i,k})_{1\leq k\leq N}$ are orthonormal symlet wavelet basis vectors and $\phi_{i,k} = \mu_{i,k}|\cdot|$. [5] This can be formulated into our optimisation algorithm's input as follows. Let $\mathcal{H}_i = \mathbb{R}^N$, $\mathcal{G}_i = \mathbb{R}^N$, $f_i(x_i) = |\langle\ (\mu_{i,k})_{k\in N}|\mathbf{DWT}(x_i)\rangle|$ for $1 \leq i \leq M$, and

$$g_k(y_k) = \begin{cases} \frac{1}{2\sigma^2}||y_k - z_i||^2 & \text{for } 1 \leq k \leq M, \\ \frac{\vartheta}{2}||y_k||^2 & \text{for } M+1 \leq k \leq 2M-1, \end{cases}$$

where **DWT** is the Discrete Wavelet Transform Operator. The proximal operator for f is computed as $\mathbf{IDWT}\left(\text{prox}_{\|\cdot\|_1}(\mathbf{DWT}(x))\right)$, where $\text{prox}_{\|\cdot\|_1}$ denotes the soft-thresholding operation and **IDWT** represents the inverse discrete wavelet transform.

Let $\mathcal{L}$ be the degradation operator and D be the disparity matrix. Then,

$$\text{for } 1 \leq k \leq M,\ L_{ki} = \begin{cases} \mathcal{L} & \text{if } i = k \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

$$\text{for } M+1 \leq k \leq 2M-1,\ L_{ki} = \begin{cases} \text{Id} & \text{if } i = k \\ -D_{i-k} & \text{if } i = k+1 \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

[5] In our experiment, we choose $\mu_{i,k} = 1$.

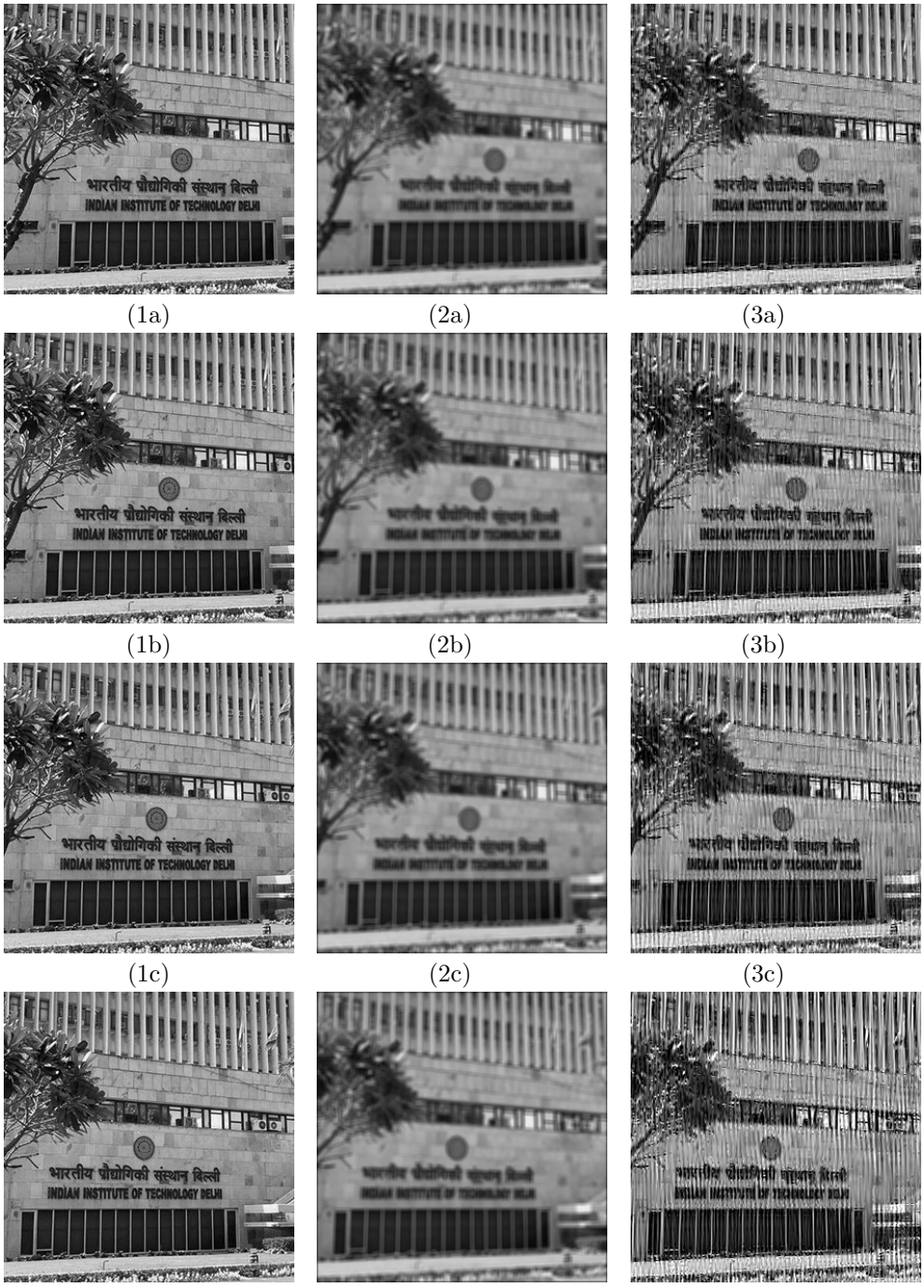

Figure 1: Original stereoscopic images (column 1), Degraded stereoscopic images (column 2) and their corresponding restored images (column 3).

We ran our algorithm for 300 iterations over the formulation described by (8). The images in Image 1 were taken after roughly equal horizontal displacement. The disparity matrix for each pair was thus found by trial and error.

4.3. Experiment 1: Synchrony versus asynchrony

We performed an experiment to determine if asynchrony can provide a speedup in practice. We also compared the performance of the asynchronous run of our implementation over variable number of processors.

4.3.1. Setup

We ran the algorithm on Problem 1 (Section 4.1) and set D to 0 for the synchronous case and 5 for asynchronous. This variable describes the maximum delay (in terms of number of iterations) that is allowed before prox evaluation, i.e., the heaviest part of our computation, is completed.
To simulate latencies, an artificial delay t associated with prox computation was added with $t \sim \mathcal{U}(0, 0.5)$ and added variance $e \sim \mathcal{N}(0, 0.005)$. We ran the algorithm for 10 iterations (which was sufficient to get $\frac{||x_n - x_{n-1}||^2}{||x_n - x_0||^2} \approx 10^{-2}$) with blocks $I_n = [1, m]$, i.e., full activation and $\mu_i = 0.42 - 0.01i$, $\gamma_i = 1$. Results were averaged over 10 runs with x_∞ being taken as the position reached value after double the total iterations.

4.3.2. Results

P	D	**Sync/Async**	$\frac{\|\|f(x_{\min}) - f(x_\infty)\|\|^2}{\|\|f(x_0) - f(x_\infty)\|\|^2}$	$\frac{\|\|x - x_\infty\|\|^2}{\|\|x_0 - x_\infty\|\|^2}$	**Time**
2	0	Sync	0.00149	0.500	185.971s
2	5	Async	0.00131	0.348	138.484s
3	0	Sync	0.00173	0.441	232.469s
3	5	Async	0.00142	0.362	130.118s

Table 1: Comparison of Asynchronous vs Synchronous run

Here, **P** is the number of processors used and D is the maximum allowed delay (see Assumption 2.2). We see from Table 1 that the Asynchronous run takes considerably less time while also approaching optimality just as well. Table 2 presents the expected pattern of the optimal number of processors that minimizes the time taken, beyond which the overhead from additional processors begins to increase the total execution time. We expect this trend

P	D	**Sync/Async**	$\frac{\|\|f(x_{\min})-f(x_\infty)\|\|^2}{\|\|f(x_0)-f(x_\infty)\|\|^2}$	$\frac{\|\|x-x_\infty\|\|^2}{\|\|x_0-x_\infty\|\|^2}$	**Time**
2	5	Async	0.00131	0.348	138.484s
3	5	Async	0.00142	0.342	130.118s
4	5	Async	0.00154	0.349	132.552s
5	5	Async	0.00193	0.357	134.044s
6	5	Async	0.00142	0.361	135.634s
7	5	Async	0.00142	0.365	139.380s
8	5	Async	0.00142	0.363	144.286s

Table 2: Time taken for Async run with different number of Processors

to remain true in general, although the ideal number of processors will change depending on the overhead due to parallelization, which is heavily dependent upon the problem and machine architecture.

4.4. Experiment 2: Hyperparameter search

In this experiment, we try to find the best performing settings for the hyperparameters γ_n and μ_n where n is the number of iterations. For a single iteration, their values were taken to be constant for different Proximal operators.

We first search to find the ideal value of $\mu_n = c_1$ and $\gamma_n = c_2$, i.e., constant w.r.t. n. This was done by performing a grid search over the logarithmic scale followed by a ternary search between the two best performing orders. The objective values for comparison were the function values for Problem 4.1 and Problem 4.2 as two separate independent searches. We found that the values: ($\mu_n = 0.332$, $\gamma_n = 0.0001$), were the optimal values for Problem 4.1 for full activation with variation in μ_n being almost the sole contributor towards the objective value reached over 10 iterations. For 0.1 activation, the values were ($\mu_n = 0.352$, $\gamma_n = 0.0001$), i.e., almost entirely same. Problem 4.2, run over 300 iterations showed the ideal constant values being of the order ($\mu_n = 0.1$, $\gamma_n = 0.0001$) for various activations, with μ_n, again being the dominant factor. A consistent trend showed the order of $\mu_n = 10^{-1}$ yielded good results. Although we conducted our grid search on the synchronous case, we found that in practice it yields similar performance improvements for asynchronous algorithms as well, so we use the same hyperparameters for both.

Next, we also compared the performance of different variations over n,

specifically μ_n, γ_n of the form -

(i) linear decrease $(a - bn)$
(ii) constant (c)
(iii) non-linear decrease $(a - \frac{b}{n})$
(iv) uniform random $(\sim U\left[\epsilon, \frac{1}{\epsilon}\right])$

Results were compiled with the same metrics as described above and they showed that linear decrease (i) performed the best with hyperparameter values $\mu_n = \max(0.01, 0.42 - 0.03n)$ for Problem-4.1. This was followed by non-linear decrease (iii) at a close second. Our findings suggest that decreasing strategies for hyperparameters tend to outperform constant ones, while increasing strategies perform poorly.

5. Data Statement

The data (images) used to replicate these experiments is publically available and can be found at `https://github.com/zevwoodstock/AsyncProx/`.

References

[1] F. Bach, R. Jenatton, J. Mairal, G. Obozinski, 2012. doi:10.1561/2200000015.

[2] A. Chambolle, T. Pock, An introduction to continuous optimization for imaging, Acta Numerica 25 (2016) 161–319. doi:10.1017/S096249291600009X.

[3] P. L. Combettes, J.-C. Pesquet, Proximal Splitting Methods in Signal Processing, Springer New York, New York, NY, 2011, pp. 185–212. doi:10.1007/978-1-4419-9569-8_10.
URL https://doi.org/10.1007/978-1-4419-9569-8_10

[4] M. Hintermüller, G. Stadler, An infeasible primal-dual algorithm for total bounded variation–based inf-convolution-type image restoration, SIAM Journal on Scientific Computing 28 (1) (2006) 1–23. arXiv:https://doi.org/10.1137/040613263, doi:10.1137/040613263.
URL https://doi.org/10.1137/040613263

[5] J. Eckstein, B. F. Svaiter, A family of projective splitting methods for the sum of two maximal monotone operators, Mathematical Programming 111 (2008) 173–199.

[6] P. R. Johnstone, J. Eckstein, Convergence rates for projective splitting, SIAM Journal on Optimization 29 (3) (2019) 1931–1957.

[7] P. R. Johnstone, J. Eckstein, Single-forward-step projective splitting: exploiting cocoercivity, Computational Optimization and Applications 78 (1) (2021) 125–166.

[8] P. R. Johnstone, J. Eckstein, Projective splitting with forward steps, Mathematical Programming (2022) 1–40.

[9] J. Eckstein, A simplified form of block-iterative operator splitting and an asynchronous algorithm resembling the multi-block alternating direction method of multipliers, Journal of Optimization Theory and Applications 173 (1) (2017) 155–182.

[10] P. L. Combettes, J. Eckstein, Asynchronous block-iterative primal-dual decomposition methods for monotone inclusions, Mathematical Programming 168 (2018) 645–672.

[11] M. N. Bui, P. L. Combettes, Z. C. Woodstock, Block-activated algorithms for multicomponent fully nonsmooth minimization, in: ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, 2022, pp. 5428–5432.

[12] J. Eckstein, J.-P. Watson, D. L. Woodruff, Projective hedging algorithms for multistage stochastic programming, supporting distributed and asynchronous implementation, Operations Research (2023).

[13] P. L. Combettes, J.-C. Pesquet, Stochastic quasi-fejér block-coordinate fixed point iterations with random sweeping, SIAM Journal on Optimization 25 (2) (2015) 1221–1248.

[14] H. H. Bauschke, P. L. Combettes, Convex Analysis and Monotone Operator Theory in Hilbert Spaces, Springer Science+Business Media, 2011.

[15] B. Martinet, Régularisation d'inéquations variationnelles par approximations successives, Revue Française D'automatique, Informatique, Recherche Opérationnelle 3 (1970) 154–158.

[16] P. L. Combettes, A. M. McDonald, C. A. Micchelli, M. Pontil, Learning with optimal interpolation norms, Numerical Algorithms 81 (2019) 695–717.

[17] P. L. Combettes, L. E. Glaudin, Proximal activation of smooth functions in splitting algorithms for convex image recovery, SIAM Journal on Imaging Sciences 12 (2019) 1905–1935.

Acknowledgments: This research was partially supported by the DFG Cluster of Excellence MATH+ (EXC-2046/1, project id 390685689) funded by the Deutsche Forschungsgemeinschaft (DFG) as well as the Research Campus MODAL funded by the German Federal Ministry of Education and Research (BMBF) (fund numbers 05M14ZAM, 05M20ZBM).

Conflict of interest statement: We declare no conflict of interest.

Author Credit Statement: This work was predominantly carried out by US, KG, and AD (with contribution level in that order) under the primary supervision of ZW and secondary supervision of SP.