# A note on asynchronous Projective Splitting in Julia

Utkarsh Sharma[1], Kashish Goel[1], Aryan Dua[1],
Project advisors: Zev Woodstock[2,3], Sebastian Pokutta[2]

**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, as surveyed by Bach et al. [1], Chambolle and Pock [2], Combettes and Pesquet [3], and Hintermüller and Stadler [4]. Despite the critical importance of first-order nonsmooth optimization, the

[1]Department of Computer Science and Engineering, I.I.T. Delhi, Hauz Khas, New Delhi 110016, India.

[2]Department of AI in Society, Science, and Technology, Zuse Institute Berlin, Takustr. 7, 12045 Berlin, Germany.

[3]Department of Mathematics and Statistics, James Madison University, MSC 1911, 60 Bluestone Dr., Harrisonburg, Virginia 22807, USA.

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 [5]. 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, with a primary application in minimizing a sum of nonsmooth convex functions [6]. In recent years, several extensions of the PS framework have been contributed by Johnstone, Eckstein, and Combettes [7, 8, 9, 10, 11]. Perhaps one of the first major theoretical breakthroughs for this class of algorithms came in 2016–2018, when Eckstein [10] and Combettes and Eckstein [11] proved that a PS algorithm will still converge when its updates are performed in parallel with bounded asynchrony. This potential for asynchronous parallelization makes PS algorithms unique among the class of provenly-convergent algorithms for solving nonsmooth convex minimization problems with a variety of benefits, as further explained in Bui, Combettes, and Woodstock [5]. Although the PS algorithm of Combettes and Eckstein [11] for convex minimization has been used to solve a variety of problems in image processing, recommender training, and progressive hedging, all implementations to-date relied on *synchronous* use of processors [5, 9, 12]. While the potential for asynchrony has been known for several years, to-date there appears to be no asynchronously parallel implementation of [11]. Furthermore, even though PS is proven to converge for a wide range of hyperparameters [10, 11], their choice can drastically influence performance in practice, and there is a lack of literature discussing their selection.

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 [11].

### *1.1. Literature Review*

One computational study was performed by [5], where it was established that the synchronous algorithms [13] and [11] (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, [11] appears to out-perform [13] in terms of wall-clock time. However, the experiments in [5] were entirely for the synchronous version of the projective splitting algorithm in [11]. A more recent variant of projective splitting is also proven to allow for parallel block-asynchronous updates [9]. While the article includes very impressive experiments, it again does not study the impact of asynchrony.

The progressive hedging method studied in [12], which is a special case of [10, 11], is shown to perform exceedingly well for this particular application. This work is encouraging for us to develop a general software implementation of [11] *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 $\operatorname{Prox}_{\Sigma f_i}$, i.e., the proximity operator for the sum $\Sigma f_i$, is oftentimes computationally expensive or intractable; on the other hand, evaluating the individual operators $(\operatorname{Prox}_{f_i})_{i \in I}$ is far easier. Minimising sums via the use of the individual proximity operators is called *splitting*.

We implement Algorithm 4 of [11] which, in addition to being a splitting algorithm, is also block-activated, and asynchronous. For minimising the sum of functions $\Sigma f_i$ for $i \in \{1, ..., m\}$, computing $\operatorname{Prox}_{\gamma f_i}$ 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 $n^{\text{th}}$ iteration. Asynchrony in our algorithm allows us to compute proximity operators $\operatorname{Prox}_{\gamma f_i}$ 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 $\bigtimes_{i=1}^{m} \mathcal{H}_i = \mathcal{H}_1 \times \ldots \times \mathcal{H}_m$, 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 $(x_i)_{i=1}^m, (y_i)_{i=1}^m \in \bigtimes_{i=1}^m \mathcal{H}_i$. 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 [11] minimises an objective function of the following form

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

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 [11].

**Algorithm 1** Combettes-Eckstein Algorithm [11]

**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**
    $\pi_n = \sum_{i=1}^{m} \big(\langle x_{i,n} \mid t^*_{i,n}\rangle - \langle a_{i,n} \mid a^*_{i,n}\rangle\big) + \sum_{k=1}^{p} \big(\big\langle t_{k,n} \mid v^*_{k,n}\big\rangle - \big\langle b_{k,n} \mid b^*_{k,n}\big\rangle\big)$
  **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**

**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 $\infty$. 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. In addition, (iv) ensures that in any current iteration, no prox computation that is left unfinished is older than $D$.

**Theorem 2.3 ([11, 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).*

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

## 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 workers and cyclically assigning prox computations over them.

### *3.1. Inputs and problem description*

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) $L$ : `AbstractMatrix` 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) $L^*$ : `AbstractMatrix` 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 $n^{\text{th}}$ 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 $x \in \bigtimes_{i=1}^{m} \mathcal{H}_i$, the input can belong to $\bigtimes_{i=1}^{m} \mathbb{R}^{k_i}$ where $k_i$ can be all different. The $L$ matrix consisting of $L_{ki}$ operators mentioned in the original equation, can be input as a matrix of operators of the form of 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

[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/`

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 : $I$ is partitioned into $M$ subsets of (approximately) equal cardinality.

*3.2. Implementation*

The algorithm is implemented in Julia using standard multi-process parallelism. On the 24-core machine, we run Julia with `-p N` flags, creating (`N`) workers plus a master process; the operating system maps these workers onto the physical cores with a one-to-one relationship.[5] All algorithm parameters are broadcast to every worker at startup, so each worker can independently execute proximal steps when requested. For each $i \in I_n$ and $k \in K_n$, the proximal subproblems appearing in Algorithm 1 are evaluated asynchronously on different workers. Concretely, at each iteration $n$, we form the shifted arguments $x^*_{i,c_i(n)}$ and $y^*_{k,d_k(n)}$, and submit the evaluations of $\mathrm{Prox}_{\gamma_{i,c_i(n)}f_i}(x^*_{i,c_i(n)})$ and $\mathrm{Prox}_{\mu_{k,d_k(n)}g_k}(y^*_{k,d_k(n)})$ as remote tasks, spread evenly across the workers.

The implementation maintains, for every outstanding proximal task, the index of the corresponding operator and the iteration at which it was launched. A user-defined integer parameter plays the role of the (iteration) delay bound $D$ in Assumption 2.2. At iteration $n$, any task whose launch iteration is older than $n-D$ is waited on (via a blocking `fetch`), and its output is immediately used to update the associated pairs $(a_{i,\cdot}, a^*_{i,\cdot})$ or $(b_{k,\cdot}, b^*_{k,\cdot})$. This update also includes tasks (with birth iterations older than the maximum delay) that did not need to be waited on. These updated values are then used to form the projective-splitting residuals

$$t_{k,n} = b_{k,n} - \sum_{i=1}^{m} L_{k,i}a_{i,n}, \qquad t^*_{i,n} = a^*_{i,n} + \sum_{k=1}^{p} L^*_{k,i}b^*_{k,n},$$

[5]If there are fewer cores, the workers will by default time-slice on the available cores. This can be avoided by launching more than 1 thread per core using `JULIA_NUM_THREADS` to ensure each worker is mapped to a single thread.

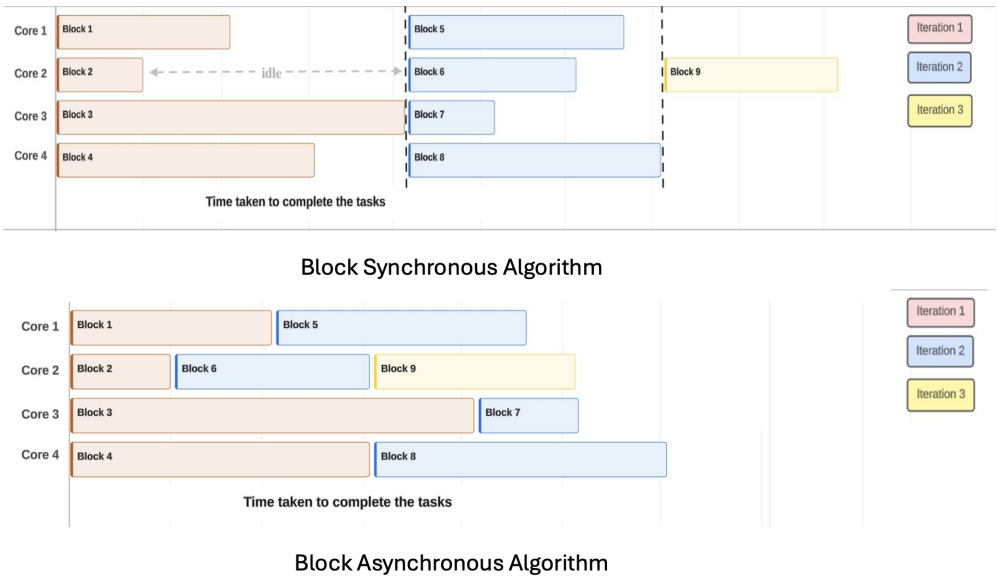


Figure 1: Schematic comparison of block-synchronous (top) and block-asynchronous (bottom) execution on four cores. Each coloured segment corresponds to a block of operators; colours indicate the iteration in which the block is logically associated. Vertical dashed lines mark synchronous iteration boundaries

and the corresponding scalars $\tau_n$, $\pi_n$ and $\theta_n$ in Algorithm 1. Tasks whose results have been harvested in this way are removed from the pool of outstanding tasks.

### *3.3. Block asynchrony*

To illustrate the difference between a block-synchronous and a block-asynchronous realisation of the algorithm, Figure 1 shows a schematic timeline on four cores. In the synchronous variant, all proximal evaluations associated with a given iteration must complete before the next iteration can begin, so slow operators delay the entire system. Our implementation is a block-asynchronous variant, since once an operator in a block finishes its proximal computation, the corresponding core can immediately start work on a block from a later iteration, while still respecting the iteration-based delay bound $D$. This allows us to overlap work across iterations and improve utilisation of the available cores.

*3.4. Recording*

We also provide mechanisms to record observations such as $||x_n - x_{n-1}||$, $||x_n - x_{\text{final}}||$[6], function values and other customisable metrics, over any of the following:

(i) epochs (refer Remark 3.1 for definition)
(ii) iterations
(iii) prox-calls.

These quantities are used to estimate the optimality of the current iterate, speed of convergence and compare relative optimality respectively over different variables.

**Remark 3.1.** *In the context of projective splitting algorithms [5], an* epoch *is a collection of consecutive iterations during which the proximal operator associated with every function in $(f_i)_{i\in I}$ and $(g_k)_{k\in K}$ is evaluated 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 single compute node equipped with dual Intel Xeon Platinum 8358 processors (Ice Lake architecture), providing a total of 64 cores ($2 \times 32$ cores at 2.6 GHz) and 256 GB of RAM. Jobs were managed using the Slurm workload manager on a Linux-based operating system. 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 sparse linear classification problem studied in [16], commonly referred to as the *latent group lasso* (or lasso with overlapping groups), and later reused in subsequent works such as [5]. The experiment in this section closely follows these prior studies and is included here to illustrate how a standard benchmark problem can be formulated within our projective splitting framework; for full modelling details, we refer the reader to the original references.

[6] $x_{\text{final}}$ refers to the position at the final iteration

Let $\{G_1, \dots, G_m\}$ be a covering of $\{1, \dots, d\}$, and define

$$X = \{(x_1, \dots, x_m) \mid x_i \in \mathbb{R}^d,\ \operatorname{supp}(x_i) \subseteq G_i\}.$$

The target classification vector is $\tilde{y} = \sum_{i=1}^m x_i$, where $(x_i)_{i=1}^m$ solves

$$\min_{(x_1,\dots,x_m)\in X} \sum_{i=1}^m \|x_i\|^2 + \sum_{k=1}^p h_k\left(\sum_{i=1}^m \langle x_i, \mu_k\rangle\right), \tag{6}$$

with $\mu_k \in \mathbb{R}^d$ and $h_k(\xi) = 10\max\{0, 1-\beta_k\xi\}$. Here $\beta_k = \omega_k \operatorname{sign}(\langle y, \mu_k\rangle)$, where $y \in \mathbb{R}^d$ is the ground-truth vector ($d = 10{,}000$). The vectors $\mu_k$ are generated by sampling $v \sim \mathcal{N}(0, I_d)$ and setting $\mu_k = v/\|v\|_2$, and the signs $\omega_k \in \{-1, 1\}$ are chosen so that 25% of the measurements are misclassified. We set $p = 1000$ measurements and $m = 1429$ groups, where each $G_i$ has cardinality 10 and overlaps with $G_{i+1}$ on 3 indices.

Since each $x_i$ is supported on $G_i$, we equivalently work with compressed variables $\tilde{x}_i \in \mathbb{R}^{10}$ and a fixed linear $F : \mathbb{R}^{10} \to \mathbb{R}^d$ that pads $\tilde{x}_i$ with zeros outside $G_i$. With this notation, (6) can be written in the form of (1) as

$$\operatorname*{arg\,min}_{\tilde{x}_1,\dots,\tilde{x}_m} \sum_{i=1}^m \|\tilde{x}_i\|^2 + \sum_{k=1}^p g_k\Big(\sum_{i=1}^m L_{ki}\tilde{x}_i\Big), \tag{7}$$

where $L_{ki} = F$ and $g_k(y_k) = 10\max\{0, 1-\beta_k\langle y_k, \mu_k\rangle\}$ with $\tilde{x}_i \in \mathbb{R}^{10}$, $\mu_k \in \mathbb{R}^d$, and $y_k = \sum_i L_{ki}\tilde{x}_i$. The corresponding proximal functions are $f_i : x_i \mapsto \|x_i\|^2$ and $g_k : y_k \mapsto 10\max\{0, 1-\beta_k\langle y_k, \mu_k\rangle\}$, and the adjoint of $L_{ki}$ is $L_{ki}^* : \mathbb{R}^d \to \mathbb{R}^{10}$ with $L_{ki}^*(y) = (y_{7i+1}, \dots, y_{7i+10})$.

The operators $\operatorname{Prox}_{f_i}$ are available in `ProximalOperators.jl`, and $\operatorname{Prox}_{g_k}$ 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 $\{x_i\}_{1\le i\le M}, x_i \in \mathbb{R}^N$, acquired sequentially by an observer moving in a fixed direction. 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 $5\times5$ averaging kernel with equal weights, and $w_i$ is additive Gaussian noise with mean zero and variance $\sigma^2 = 0.0001$.

We define linear shift operators $D_i : \mathbb{R}^N \to \mathbb{R}^N, i \in \{1, \ldots, M-1\}$, that model the geometric relationship between successive views. Each $D_i$ applies a horizontal shift (warp) to the pixels of an image represented as a vector in $\mathbb{R}^N$. The stereoscopy $x_i \approx D_i x_{i+1}$ is modelled by these successive shift operators, with the shift amounts estimated by us before the experiment is run.[7] 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 $\phi_{i,k} = \mu_{i,k}| \cdot |$.[8] [9] 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 linear "shift" operator as defined in (8). Then,

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

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

[7] We note that the shift operator can also induce some errors, since a horizontal shift of pixels is not the same thing as shifting perspective in a 3d environment.

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

[9] Gaussian blurring is a standard degradation model in image recovery, although in practical applications the blur operator typically needs to be estimated.

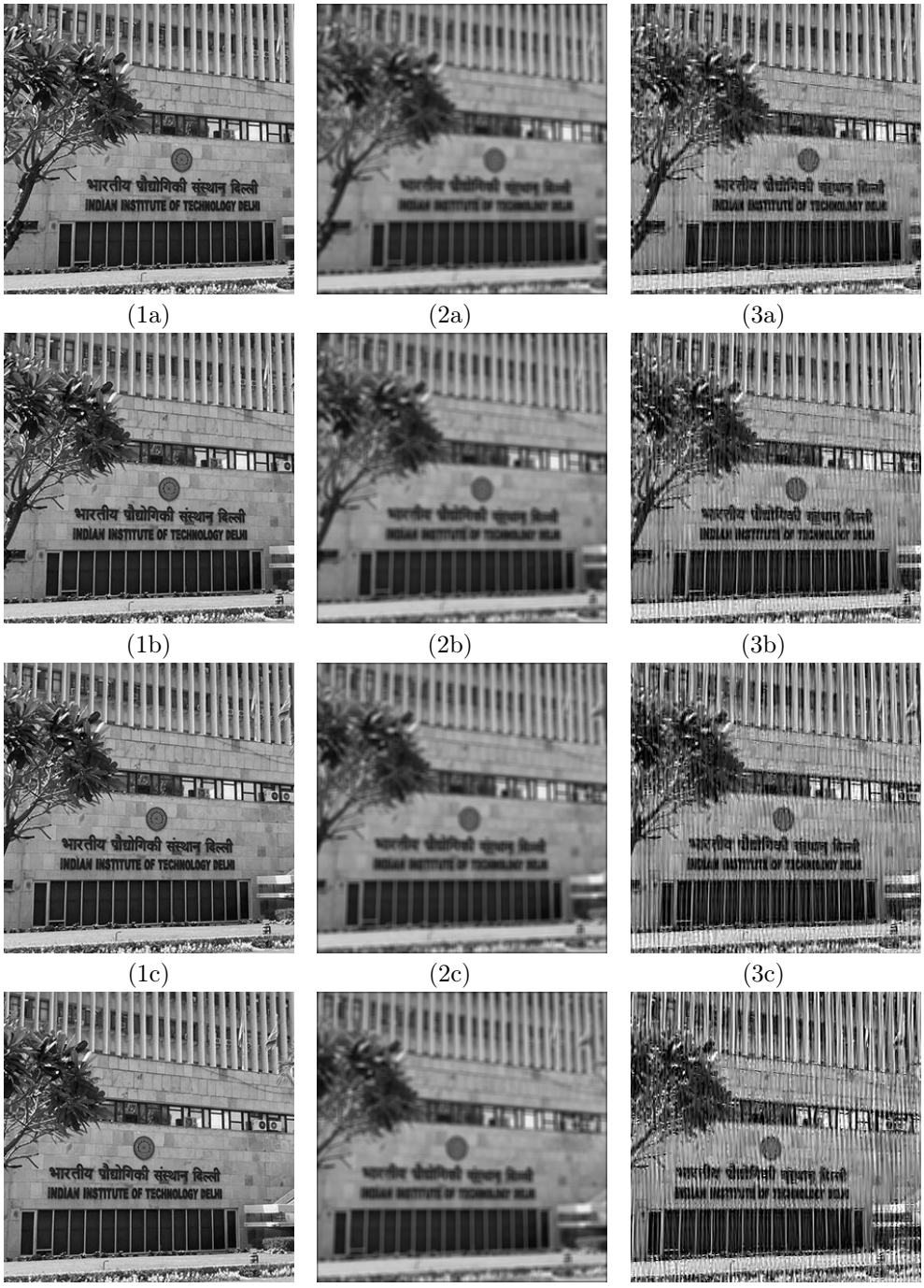


Figure 2: Original stereoscopic images (column 1), Degraded stereoscopic images (column 2) and their corresponding restored images (column 3).

We ran our algorithm for 300 iterations over the formulation described by (8). The images in Image 2 were taken after roughly equal horizontal displacement. The shift operator 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 process workers (or processes).

#### *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 $x_\infty$ being taken as the position reached in the first iteration $n$ with relative error $(\frac{||x_n - x_{n-1}||^2}{||x_n - x_0||^2}) < 10^{-3}$.

#### *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 | 0.00173 | 0.598 | 881.144s |
| 2 | 5 | Async | 0.00129 | 0.335 | 728.214s |
| 3 | 0 | Sync | 0.00181 | 0.641 | 923.442s |
| 3 | 5 | Async | 0.00132 | 0.376 | 701.419s |

Table 1: Comparison of Asynchronous vs Synchronous run

Here, **P** is the number of processes spawned (each, on a separate core) and $D$ is the maximum allowed delay (see Assumption 2.2). Note that $D = 0$ is akin to a block-synchronous run since after each iteration, we would wait for all workers launched in the previous iteration to finish. We see from Table 1 that the Asynchronous run takes considerably less time while also

| **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 | 0.00129 | 0.335 | 728.214s |
| 3 | 5 | Async | 0.00132 | 0.376 | 701.419s |
| 4 | 5 | Async | 0.00143 | 0.339 | 745.372s |
| 5 | 5 | Async | 0.00183 | 0.359 | 747.237s |
| 6 | 5 | Async | 0.00139 | 0.341 | 753.665s |
| 7 | 5 | Async | 0.00139 | 0.345 | 826.691s |
| 8 | 5 | Async | 0.00177 | 0.428 | 912.192s |

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

approaching optimality just as well. Table 2 compares the performance across the number of processes launched which seems to reach a minimum at 3, beyond which the overhead from additional processes begins to increase the total execution time. We expect this trend to remain true in general, although the actual number of ideal processes will vary depending on the exact problem and the underlying hardware architecture.

### *4.4. Experiment 2: Hyperparameter search*

In this experiment, we try to find the best performing settings for the hyperparameters $\gamma_n$ and $\mu_n$ 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 $\mu_n$ 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 $\mu_n$, 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 $\mu_n, \gamma_n$ 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.

### *4.5. Limitations and future work*

In these experiments, variability in proximal evaluation times was introduced via artificial random delays in order to emulate heterogeneous computation costs across operators. While this allows us to study the qualitative impact of asynchrony in a controlled setting, experiments arising from applications with naturally uneven proximal runtimes—such as distributed environments where proximal operators are evaluated on remote or heterogeneous compute resources—would be more compelling. Investigating such real-world settings, where delays arise organically rather than synthetically, is an important direction for future work. Extending the experimental evaluation in 4.1 to real-world datasets and to applications whose problem structure or data distributions differ substantially from the synthetic setting is also an important direction for future work. Additionally, the extension to $M$ images in 4.2 could be more suitably used for deblurring a burst of images, such as consecutive video frames captured while moving in the same direction (e.g., video recorded from a vehicle).

## 5. Conclusion

We presented an open-source Julia implementation of the asynchronous projective splitting algorithm of Combettes and Eckstein for fully nonsmooth convex optimization. Our experiments on sparse classifier training

and stereoscopic image recovery demonstrate that the algorithm works in practice. We verified that asynchrony can reduce wall-clock time while preserving convergence quality. We also observed that hyperparameter scheduling, especially for $\mu_n$, strongly affects performance, with decreasing strategies outperforming constant ones.

This work bridges a gap between theoretical guarantees of asynchronous projective splitting and practical implementations.

## 6. 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] 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.

[6] 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.

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

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

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

[10] 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.

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

[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). The work of Z. Woodstock was partially supported by the National Science Foundation under grant DMS-2532423.

We thank the anonymous referee for their helpful feedback.

**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.