Algebraic characterization of equivalence between optimization algorithms

Laurent Lessard[1*] and Madeleine Udell[2]

[1]Northeastern University.
[2]Stanford University.

*Corresponding author(s). E-mail(s): l.lessard@northeastern.edu;
Contributing authors: udell@stanford.edu;

Abstract

When are two algorithms the same? How can we be sure a recently proposed algorithm is novel, and not a minor twist on an existing method? In this paper, we present a framework for reasoning about equivalence between a broad class of iterative algorithms, with a focus on algorithms designed for convex optimization. We propose several notions of what it means for two algorithms to be equivalent, and provide computationally tractable means to detect equivalence. Our main definition, oracle equivalence, states that two algorithms are equivalent if they result in the same sequence of calls to the function oracles (for suitable initialization). Borrowing from control theory, we use state-space realizations to represent algorithms and characterize algorithm equivalence via transfer functions. Our framework can also identify and characterize equivalence between algorithms that use different oracles that are related via a linear fractional transformation. Prominent examples include linear transformations and function conjugation.

Keywords: optimization algorithm, algorithm equivalence, algorithm transformation

1 Introduction

Large-scale optimization problems in machine learning, signal processing, and imaging have fueled ongoing interest in iterative optimization algorithms. New optimization algorithms are regularly proposed to capture more complicated models, reduce computational burdens, or obtain stronger performance and convergence guarantees.

However, the *novelty* of an algorithm can be difficult to establish because algorithms can be written in different equivalent forms. For example, Algorithm 1.1 was originally proposed by Popov [1] in the context of solving saddle point problems. This method was later generalized by Chiang et al. [2, §4.1] in the context of online optimization. Algorithm 1.2 is a reformulation of Algorithm 1.1 adapted for use in generative adversarial networks (GANs) [3]. Algorithm 1.3 is an adaptation of *Optimistic Mirror Descent* [4] used by Daskalakis et al. [5] and also used to train GANs. Finally, Algorithm 1.4 was proposed by Malitsky [6] for solving monotone variational inequality problems. In all four algorithms, the vectors x_1^k and x_2^k are algorithm states, η is a tunable parameter, and F is the gradient of the loss function.

Algo. 1.1 (Modified Arrow–Hurwicz)

for $k = 0, 1, 2, \dots$ **do**
$x_1^{k+1} = x_1^k - \eta F(x_2^k)$
$x_2^{k+1} = x_1^{k+1} - \eta F(x_2^k)$
end for

Algo. 1.2 (Extrapolation from the past)

for $k = 0, 1, 2, \dots$ **do**
$x_2^k = x_1^k - \eta F(x_2^{k-1})$
$x_1^{k+1} = x_1^k - \eta F(x_2^k)$
end for

Algo. 1.3 (Optimistic Mirror Descent)

for $k = 0, 1, 2, \dots$ **do**
$x_2^{k+1} = x_2^k - 2\eta F(x_2^k) + \eta F(x_2^{k-1})$
end for

Algo. 1.4 (Reflected Gradient Method)

for $k = 0, 1, 2, \dots$ **do**
$x_1^{k+1} = x_1^k - \eta F(2x_1^k - x_1^{k-1})$
end for

Algorithms 1.1–1.4 are equivalent in the sense that when suitably initialized, the sequences $(x_1^k)_{k\geq 0}$ and $(x_2^k)_{k\geq 0}$ are identical for all four algorithms.[1] Although these particular equivalences are not difficult to verify and many have been explicitly pointed out in the literature, for example in [3], algorithm equivalence is not always immediately apparent.

In this paper, we present a framework for reasoning about algorithm equivalence, with the ultimate goal of making the analysis and design of algorithms more principled and streamlined. This includes:

- A universal way of representing algorithms, inspired by methods from control theory.
- Sensible definitions of what it means for algorithms to be equivalent.
- A computationally efficient way to verify whether two algorithms are equivalent.

Briefly, our method is to parse each algorithm to a standard form as a linear system in feedback with a nonlinearity; to compute the *transfer function* of each linear system; and to check whether certain key relationships hold between the transfer functions of the algorithms in question.

This paper is organized as follows. In Section 2, we briefly summarize existing literature related to our work. In Section 3, we introduce four examples of equivalent algorithms that motivate our framework. In Section 4, we briefly review important background on linear systems and optimization used throughout the paper and in

[1] In their original formulations, Algorithms 1.1, 1.2 and 1.4 included projections onto convex constraint sets. We assume an unconstrained setting here for illustrative purposes. Some of the equivalences no longer hold in the constrained case.

Section 5, we present our control-inspired mathematical framework for algorithm representation. We formally define three notions of algorithm equivalence: *oracle equivalence* (Section 6), *shift equivalence* (Section 7), and *LFT equivalence* (Section 8) to handle cases with: one oracle, multiple oracles, and different but related oracles, respectively. We discuss further generalizations and applications in Section 9 and conclude in Section 10.

2 Related work

Within the optimization literature, several standard forms have been proposed to represent problems and algorithms. For example, the CVX* modeling languages represent (disciplined) convex optimization problems in a standard conic form, building up the representations of complex problems from a few basic functions and a small set of composition rules [7–11]. This paper builds on a foundation developed by Lessard et al. [12] that represents first-order algorithms as linear systems in feedback with a nonlinearity. Lessard et al. use this representation to analyze convergence properties of an algorithm with integral quadratic constraints. Our work generalizes their representation to algorithms that use multiple related oracles.

There are rich connections between many first-order methods for convex optimization. These algorithms are surveyed in a recent textbook by Ryu and Yin, which summarizes and unifies several operator splitting methods for convex optimization [13]. Many of these connections are well known to experts, but the connections have traditionally been complex to explain, communicate, or even remember. For example, Boyd et al. [14] write, "There are also a number of other algorithms distinct from but inspired by ADMM. For instance, Fukushima [15] applies ADMM to a dual problem formulation, yielding a 'dual ADMM' algorithm, which is shown in [16] to be equivalent to the 'primal Douglas–Rachford' method discussed in [17, §3.5.6]." As another example, Chambolle and Pock in [18] proposed a new primal-dual splitting algorithm and demonstrated that transformations of their algorithm can yield Douglas–Rachford splitting and ADMM, using a full page of mathematics to sketch the connection. Using our framework, the relations between the Chambolle–Pock method, Douglas–Rachford splitting, and ADMM can be established precisely and automatically.

3 Motivating examples

Algorithms 1.1–1.4 discussed in Section 1 were equivalent in a strong sense; the iterates were in exact correspondence. In this paper, we adopt a broader view of equivalence, which we now illustrate with four motivating examples. Each example provides a different way that we consider two algorithms to be equivalent.

Algo. 3.1

for $k = 0, 1, 2, \ldots$ **do**
$x_1^{k+1} = 2x_1^k - x_2^k - \frac{1}{10}\nabla f(2x_1^k - x_2^k)$
$x_2^{k+1} = x_1^k$
end for

Algo. 3.2

for $k = 0, 1, 2, \ldots$ **do**
$\xi_1^{k+1} = \xi_1^k - \xi_2^k - \frac{1}{5}\nabla f(\xi_1^k)$
$\xi_2^{k+1} = \xi_2^k + \frac{1}{10}\nabla f(\xi_1^k)$
end for

First consider Algorithms 3.1 and 3.2. We may transform the iterates of Algorithm 3.1 by the invertible linear map $\xi_1^k = 2x_1^k - x_2^k, \xi_2^k = -x_1^k + x_2^k$ to yield the iterates of Algorithm 3.2. Although the iterates are not in exact correspondence as in Algorithms 1.1–1.4, the sequences $(x_1^k)_{k\geq 0}$ and $(x_2^k)_{k\geq 0}$ are equivalent to the sequences $(\xi_1^k)_{k\geq 0}$ and $(\xi_2^k)_{k\geq 0}$ up to an invertible linear transformation.

Algo. 3.3

for $k = 0, 1, 2, \ldots$ **do**
$x_1^{k+1} = 3x_1^k - 2x_2^k + \frac{1}{5}\nabla f(-x_1^k + 2x_2^k)$
$x_2^{k+1} = x_1^k$
end for

Algo. 3.4

for $k = 0, 1, 2, \ldots$ **do**
$\xi^{k+1} = \xi^k - \frac{1}{5}\nabla f(\xi^k)$
end for

The second example consists of Algorithms 3.3 and 3.4. Algorithm 3.4 is ordinary gradient descent. These algorithms do not even have the same number of state variables, so these algorithms are *not* equivalent up to an invertible linear transformation. But when suitably initialized, we may transform the iterates of Algorithm 3.3 by the linear map $\xi^k = -x_1^k + 2x_2^k$ to yield the iterates of Algorithm 3.4. This transformation is linear but not invertible. Instead, notice that the sequence of calls to the gradient oracle are identical: the algorithms satisfy *oracle equivalence*, a notion we will define formally later in this paper. Note that Algorithms 3.1 and 3.3 look similar, yet Algorithm 3.1 is *not* equivalent to gradient descent.

Algo. 3.5 (Douglas–Rachford)

for $k = 0, 1, 2, \ldots$ **do**
$x_1^{k+1} = \operatorname{prox}_f(x_3^k)$
$x_2^{k+1} = \operatorname{prox}_g(2x_1^{k+1} - x_3^k)$
$x_3^{k+1} = x_3^k + x_2^{k+1} - x_1^{k+1}$
end for

Algo. 3.6 (Simplified ADMM)

for $k = 0, 1, 2, \ldots$ **do**
$\xi_1^{k+1} = \operatorname{prox}_g(\xi_2^k - \xi_3^k)$
$\xi_2^{k+1} = \operatorname{prox}_f(\xi_1^{k+1} + \xi_3^k)$
$\xi_3^{k+1} = \xi_3^k + \xi_1^{k+1} - \xi_2^{k+1}$
end for

The third example consists of Algorithms 3.5 and 3.6. These algorithms are known as Douglas–Rachford splitting [19, 20] and a special case of the alternating direction method of multipliers (ADMM) [13, §8][14], respectively. With suitable initialization, they will generate the same sequence of calls to the proximal operators, ignoring the very first call to one of the oracles. Specifically, Algorithm 3.6 is initialized as $\xi_2^0 = x_1^1$, $\xi_3^0 = x_3^0 - x_1^1$ and the first call to prox_f in Algorithm 3.5 is ignored. We will say they are equivalent up to a prefix or shift: they satisfy *shift equivalence*. We will revisit these algorithms in Section 7.

Algo. 3.7 (Proximal gradient)

for $k = 0, 1, 2, \ldots$ **do**
$y^k = x^k - t\nabla f(x^k)$
$x^{k+1} = \operatorname{prox}_{tg}(y^k)$
end for

Algo. 3.8 (Conjugate proximal gradient)

for $k = 0, 1, 2, \ldots$ **do**
$y^k = x^k - t\nabla f(x^k)$
$x^{k+1} = y^k - t\operatorname{prox}_{\frac{1}{t}g^*}(\frac{1}{t}y^k)$
end for

Finally, consider Algorithms 3.7 and 3.8. These algorithms do not even call the same oracles; the first algorithm calls ∇f and prox_{tg} while the other calls ∇f and $\text{prox}_{\frac{1}{t}g^*}$ (the proximal operator of the Fenchel conjugate of g). Nevertheless, these two oracles are related via Moreau's identity: $x = \text{prox}_{tg}(x) + t\text{prox}_{\frac{1}{t}g^*}(\frac{1}{t}x)$ and applying this identity immediately relates Algorithms 3.7 and 3.8. These algorithms satisfy *LFT equivalence* and we will revisit them in Section 8.1.

In Sections 6–8, we will develop increasingly general notions of equivalence that cover all the motivating examples above and more. Before we can formally define algorithm equivalence, we begin by introducing the mathematical representation, borrowed from control theory, that we use to describe iterative algorithms.

4 Preliminaries

We let $\mathcal{V}$ denote a generic real vector space and $\mathcal{V}^n := \mathcal{V} \times \cdots \times \mathcal{V}$ (n times). We represent $x \in \mathcal{V}^n$ as a column vector with subvectors indexed using subscripts. In other words, $x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$ where $x_1, \ldots, x_n \in \mathcal{V}$. Superscripts are used to index sequences of vectors. For example, we may write $(x^0, x^1, \ldots)$ to denote a semi-infinite sequence of vectors with $x^k \in \mathcal{V}^n$ for each $k \geq 0$. If $A \in \mathbb{R}^{m \times n}$ and $x \in \mathcal{V}^n$, we overload matrix multiplication by writing $y = Ax \in \mathcal{V}^m$ to mean: $y_i = \sum_{j=1}^n A_{ij} x_j$ for $i = 1, \ldots, m$. In this case, we say that y is a *linear function* of x.

An *oracle* is a function $\phi : \mathcal{V} \to \mathcal{V}$. We denote the diagonal concatenation of many oracles $(\phi_1, \ldots, \phi_p)$ using an upper-case letter: $\Phi : \mathcal{V}^p \to \mathcal{V}^p$, where $u = \Phi(y)$ means that $u_i = \phi_i(y_i)$ for $i = 1, \ldots, p$.

Oracle-based iterative algorithms

We assume an oracle-based model for our iterative algorithms. The algorithm can query a set of oracles at discrete query points [21, §4][22, §1][23, §1]. For algorithms that solve optimization problems, oracles might include the gradient or proximal operator of a function, or projection onto a constraint set [24, §6][25, §2][26, §1]. We assume that the oracle outputs are unique and deterministic, possibly given some internal state such as the seed of a pseudorandom number generator. For example, a subgradient oracle might return the subgradient of minimum norm, and a stochastic gradient oracle might return the gradient corresponding to a given random seed.

For an iterative algorithm that uses oracles $(\phi_1, \ldots, \phi_p)$, we assume the following.

1. The algorithm maintains an internal *state* $x^k \in \mathcal{V}^n$ that is initialized to some x^0 before the algorithm begins.
2. During iteration k, each oracle ϕ_i is queried exactly once. We call the associated query point $y_i^k \in \mathcal{V}$ and the query result $u_i^k \in \mathcal{V}$. In other words, $u_i^k = \phi_i(y_i^k)$.
3. During iteration k, the oracles are queried in a prescribed order $\phi_{i_1}, \ldots, \phi_{i_p}$. Each query point $y_{i_j}^k$ is a *linear function* of the state x^k and possibly of the query results $u_{i_1}^k, \ldots, u_{i_{j-1}}^k$ obtained thus far.
4. Once all oracles have been queried, the internal state x^k is updated to x^{k+1} using a *linear function* of $x^k \in \mathcal{V}^n$ and of $u^k \in \mathcal{V}^p$.

5. All aforementioned linear functions are the same at every iteration (independent of k). In other words, the algorithm is *time-invariant*.

We will see that this class of algorithms includes commonly used algorithms, such as accelerated methods, proximal methods, operator splitting methods, and more [12, 27].

Our framework excludes algorithms whose parameters explicitly depend on the iterate index k, such as gradient-based methods with diminishing stepsizes. We view time-varying algorithms as schemes for switching between different time-invariant algorithms. Since our aim is to reason about algorithm equivalence, we restrict our attention to time-invariant algorithms.

Here is a pseudo-code implementation of a generic iterative algorithm that satisfies the assumptions above.

Algo. 4.1 Implementation of a generic iterative algorithm

Initialize:
  $x^0 \in \mathcal{V}^n$
for $k = 0, 1, 2, \ldots$ **do**
  for $i = 1, \ldots, p$ **do**
    $y_i^k = \sum_{j=1}^{n} C_{ij} x_j^k + \sum_{j=1}^{i-1} D_{ij} u_j^k$ ▷ Evaluate query point for i^{th} oracle.
    $u_i^k = \phi_i(y_i^k)$ ▷ Query i^{th} oracle.
  end for
  $x^{k+1} = Ax^k + Bu^k$ ▷ Update internal state.
end for

Remark 1. In Algorithm 4.1, we assumed the oracles were queried in the order $\phi_1, \ldots, \phi_p$, so the D matrix is strictly lower triangular. If the oracles were queried in a different order, the rows and columns of D would be permuted accordingly.

State-space form

We can write the updates in Algorithm 4.1 in the more compact form

$$x^{k+1} = Ax^k + Bu^k, \tag{1a}$$
$$y^k = Cx^k + Du^k, \tag{1b}$$
$$u^k = \Phi(y^k), \tag{1c}$$

where $A \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{n \times p}$, $C \in \mathbb{R}^{p \times n}$, and $D \in \mathbb{R}^{p \times p}$. The equations (1) can also be represented visually using a block diagram, as in Fig. 1.

The representation of Fig. 1 separates the *oracles*, which map $y^k \mapsto u^k$, from the *algorithm*, which maps $(x^0, u^0, u^1, \ldots, u^k) \mapsto y^k$. This decomposition was first developed in [12]. The algorithm is characterized by the matrices (A, B, C, D), which are called a *state-space realization*, and are a widely used representation for linear time-invariant dynamical systems [28].

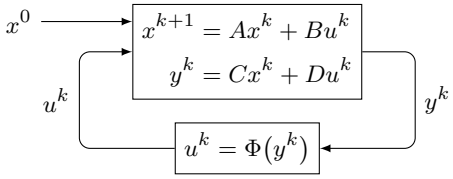

Fig. 1: Block diagram representation of a generic iterative algorithm.

Oracle normalization

We will assume without loss of generality that the oracle satisfies $\Phi(0) = 0$. This is possible because the dynamics of Eq. (1) will generally have a fixed point. That is, there exists $(x^\star, u^\star, y^\star)$ satisfying

$$x^\star = Ax^\star + Bu^\star, \tag{2a}$$
$$y^\star = Cx^\star + Du^\star, \tag{2b}$$
$$u^\star = \Phi(y^\star). \tag{2c}$$

For gradient-based algorithms for smooth unconstrained optimization, $\Phi(y) = \nabla f(y)$, where f is the objective function, so we typically have $y^\star \in \operatorname{argmin}_y f(y)$ and the fixed point is $u^\star = \nabla f(y^\star) = 0$. However, this need not always be the case. For example, in distributed optimization (see Section 6.1) the gradient is not zero at optimality so $u^\star \neq 0$. If we define new coordinates measured in relation to a fixed point, namely:

$$\tilde{x}^k := x^k - x^\star, \qquad \tilde{u}^k := u^k - u^\star, \qquad \tilde{y}^k := y^k - y^\star, \qquad \tilde{\Phi}(y) := \Phi(y + y^\star) - u^\star,$$

then combining Eqs. (1) and (2) yields the transformed dynamics

$$\tilde{x}^{k+1} = A\tilde{x}^k + B\tilde{u}^k, \tag{3a}$$
$$\tilde{y}^k = C\tilde{x}^k + D\tilde{u}^k, \tag{3b}$$
$$\tilde{u}^k = \tilde{\Phi}(\tilde{y}^k). \tag{3c}$$

Comparing Eqs. (1) and (3), we see that the state-space matrices (A, B, C, D) are unchanged, but the transformed oracle now satisfies $\tilde{\Phi}(0) = 0$.

We now present a few examples that illustrate how to find a state-space realization.

Example: Reflected Gradient Method

Consider Algorithm 1.4, which uses oracle F and has update equation

$$x_1^{k+1} = x_1^k - \eta F(2x_1^k - x_1^{k-1}). \tag{4}$$

Since the update for x_1^{k+1} depends on both x_1^k and x_1^{k-1}, we augment the internal state to include this past iterate. To this effect, we define $x_2^k := x_1^{k-1}$ and obtain update

equations with state (x_1^k, x_2^k) that only depend on the previous timestep:

$$\begin{aligned} x_1^{k+1} &= x_1^k - \eta F(2x_1^k - x_2^k) \\ x_2^{k+1} &= x_1^k \end{aligned}$$

Define the oracle query point y^k and query result u^k. We can now express Eq. (4) in the form of Algorithm 4.1 and Eq. (1):

$$\left.\begin{aligned} y^k &= 2x_1^k - x_2^k \\ u^k &= F(y^k) \end{aligned}\right\} \qquad y^k = \begin{bmatrix} 2 & -1 \end{bmatrix} \begin{bmatrix} x_1^k \\ x_2^k \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} u^k$$

$$\left.\begin{aligned} x_1^{k+1} &= x_1^k - \eta F(2x_1^k - x_2^k) = x_1^k - \eta u^k \\ x_2^{k+1} &= x_1^k \end{aligned}\right\} \qquad \begin{bmatrix} x_1^{k+1} \\ x_2^{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1^k \\ x_2^k \end{bmatrix} + \begin{bmatrix} -\eta \\ 0 \end{bmatrix} u^k$$

Therefore, a state-space realization for Algorithm 1.4 is given by

$$(A, B, C, D) = \left(\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} -\eta \\ 0 \end{bmatrix}, \begin{bmatrix} 2 & -1 \end{bmatrix}, \begin{bmatrix} 0 \end{bmatrix} \right). \tag{5}$$

Example: simplified ADMM

Consider Algorithm 3.6 (simplified ADMM), which uses state variables $(\xi_1^k, \xi_2^k, \xi_3^k)$, oracles $(\mathrm{prox}_f, \mathrm{prox}_g)$, and update equations

$$\xi_1^{k+1} = \mathrm{prox}_g(\xi_2^k - \xi_3^k), \quad \xi_2^{k+1} = \mathrm{prox}_f(\xi_1^{k+1} + \xi_3^k), \quad \xi_3^{k+1} = \xi_3^k + \xi_1^{k+1} - \xi_2^{k+1}. \tag{6}$$

Define the oracle query points (y_1^k, y_2^k) and query results (u_1^k, u_2^k). We can now express Eq. (6) in the form of Algorithm 4.1 and Eq. (1):

$$\left.\begin{aligned} y_2^k &= \xi_2^k - \xi_3^k \\ u_2^k &= \mathrm{prox}_g(y_2^k) \\ y_1^k &= \xi_1^{k+1} + \xi_3^k = \xi_3^k + u_2^k \\ u_1^k &= \mathrm{prox}_f(y_1^k) \end{aligned}\right\} \qquad \begin{bmatrix} y_1^k \\ y_2^k \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} \xi_1^k \\ \xi_2^k \\ \xi_3^k \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1^k \\ u_2^k \end{bmatrix}$$

$$\left.\begin{aligned} \xi_1^{k+1} &= u_2^k \\ \xi_2^{k+1} &= u_1^k \\ \xi_3^{k+1} &= \xi_3^k + \xi_1^{k+1} - \xi_2^{k+1} = \xi_3^k + u_2^k - u_1^k \end{aligned}\right\} \qquad \begin{bmatrix} \xi_1^{k+1} \\ \xi_2^{k+1} \\ \xi_3^{k+1} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \xi_1^k \\ \xi_2^k \\ \xi_3^k \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} u_1^k \\ u_2^k \end{bmatrix}$$

Therefore, a state-space realization for Algorithm 3.6 is given by

$$(A, B, C, D) = \left(\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ -1 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \right). \tag{7}$$

Explicit and implicit implementations

Given a state-space realization (A, B, C, D) where $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times p}$, $C \in \mathbb{R}^{p\times n}$, and $D \in \mathbb{R}^{p\times p}$, when is it possible to construct a corresponding step-by-step implementation in the form of Algorithm 4.1? By Remark 1, it is possible provided there exists a permutation matrix P such that $P^\top DP$ is strictly lower-triangular. The permutation P describes the order in which the oracles $\phi_1, \ldots, \phi_p$ will be evaluated in Algorithm 4.1. Put another way, if we view D as the adjacency matrix for a directed graph, the corresponding graph should be a *directed acyclic graph* (DAG).

If the D matrix does *not* correspond to a DAG, the graph will exhibit a cycle, which will manifest itself as an implicit equation involving oracles.[2] For example, consider the realization $(A, B, C, D) = (1, -t, 1, -t)$. This algorithm has the update equation

$$\left.\begin{aligned} x^{k+1} &= x^k - tu^k \\ y^k &= x^k - tu^k \\ u^k &= \phi(y^k) \end{aligned}\right\} \quad \Longleftrightarrow \quad x^{k+1} = x^k - t\phi(x^{k+1}). \tag{8}$$

The D matrix is not strictly lower-triangular and x^{k+1} is defined *implicitly*.

Definition 1. If the state-space realization (A, B, C, D) for an algorithm has the property that there exists a permutation matrix P such that $P^\top DP$ is strictly lower-triangular, we say that the algorithm has an *explicit* implementation. Otherwise, we say it has an *implicit* implementation.

Implicit implementations do occur in practice. For example, consider Eq. (8) and let $\phi = \nabla f$, where f is convex. Then, by the first-order optimality conditions, we have

$$\begin{aligned} x^{k+1} = \operatorname{prox}_{tf}(x^k) \quad &\Longleftrightarrow \quad x^{k+1} = \operatorname*{argmin}_x \left(f(x) + \tfrac{1}{2t}\left\|x - x^k\right\|^2 \right) \\ &\Longleftrightarrow \quad x^{k+1} = x^k - t\nabla f(x^{k+1}). \end{aligned}$$

In other words, using the oracle prox_{tf} yields an explicit implementation, but using the oracle ∇f yields an implicit implementation.

State-space realizations conveniently parametrize a large class of explicit and implicit algorithms in terms of matrices (A, B, C, D), but the representation is not unique. For example, Algorithms 1.1–1.4 have different realizations (A, B, C, D) despite having identical state sequences. In the next section, we show how tools from control theory can clarify the relations between these sorts of representations.

5 Algorithm representation

In this section, we explain how to represent algorithms using *transfer functions*, a standard tool in linear systems and control theory [29, §1–3][28, §1,2,5]. We will give an overview of relevant terminology and show how to convert an algorithm to and from the transfer function representation.

[2] also known as an "algebraic loop" or a "circular dependency".

In Section 4, we discussed algorithms that have a state-space realization

$$x^{k+1} = Ax^k + Bu^k, \tag{9a}$$
$$y^k = Cx^k + Du^k. \tag{9b}$$

We represent semi-infinite sequences such as $(x^0, x^1, \dots)$ using their *z-transforms*. That is, we define the formal power series[3]

$$\mathcal{Z}[x^k] = \hat{x}(z) := \sum_{k=0}^{\infty} x^k z^{-k}$$

and similarly for $\hat{u}(z)$ and $\hat{y}(z)$. When taking the z-transform of the forward-shifted sequence $(x^1, x^2, \cdots)$, we have:

$$\mathcal{Z}\left[x^{k+1}\right] = \sum_{k=0}^{\infty} x^{k+1} z^{-k} = z(\hat{x}(z) - x^0)$$

Evaluating the z-transform of (9), we obtain:

$$z(\hat{x}(z) - x^0) = A\hat{x}(z) + B\hat{u}(z), \tag{10a}$$
$$\hat{y}(z) = C\hat{x}(z) + D\hat{u}(z). \tag{10b}$$

Eliminating $\hat{x}(z)$ from (10), we obtain

$$\hat{y}(z) = \underbrace{zC(zI - A)^{-1}}_{\hat{O}(z)} x^0 + \underbrace{\left(D + C(zI - A)^{-1}B\right)}_{\hat{H}(z)} \hat{u}(z) \tag{11}$$

The (matrix-valued) functions $\hat{O}(z)$ and $\hat{H}(z)$ are convenient to work with because they relate $\hat{y}(z)$ and $\hat{u}(z)$ via conventional matrix multiplication. The function $\hat{H}(z)$ is called the *transfer function*, and this is how we will represent the state-space system (A, B, C, D). We use the special notation

$$\hat{H}(z) = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] := D + C(zI - A)^{-1}B. \tag{12}$$

For ease of notation, we will often omit the "(z)" after each transfer function, so when we write $\hat{H}_1 = \hat{H}_2$, we mean that $\hat{H}_1(z) = \hat{H}_2(z)$ for all z. We will also overload oracles so that they may apply to the z-transforms directly by threading across coefficients. Namely, if $\Phi(y^k) = u^k$ for $k = 0, 1, \dots$, we will write:

$$\Phi(\hat{y}) := \sum_{k=0}^{\infty} \Phi(y^k) z^{-1} = \sum_{k=0}^{\infty} u^k z^{-1} = \hat{u}$$

[3]The use of z^{-1} as the variable in the z-transform is a common convention in control theory.

Therefore, the block diagram of Fig. 1 can be written in terms of transfer functions and z-transforms as in Fig. 2.

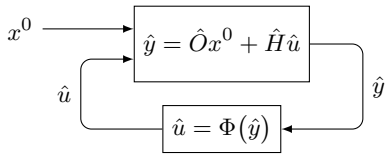

Fig. 2: Block diagram representation of a generic optimization algorithm expressed in terms of its transfer function $\hat{H}$ and the z-transforms of its inputs and outputs.

Applying the formula (12) to the state-space matrices of simplified ADMM (Algorithm 3.6) from (7), we obtain the transfer function

$$\hat{H}_{3.6} = \begin{bmatrix} \frac{-1}{z-1} & \frac{z}{z-1} \\ \frac{2z-1}{z(z-1)} & \frac{-1}{z-1} \end{bmatrix}. \tag{13}$$

The transfer function can be readily computed directly from the update equations by replacing each state by its z-transform, neglecting initial conditions, and eliminating the state variables. The following Python code computes the transfer function for Algorithm 3.6 (Eq. (13)) starting from the update equations.

```
from sympy import *

# Define the symbols
var('z xi1 xi2 xi3 y1 y2 u1 u2')

# Define the update equations
eqns = [
  Eq(z*xi1, u2),                  # xi1[k+1] = u2[k]
  Eq(z*xi2, u1),                  # xi2[k+1] = u1[k]
  Eq(z*xi3, xi3 + z*xi1 - z*xi2), # xi3[k+1] = xi3[k] + xi1[k+1] - xi2[k+1]
  Eq(y1, z*xi1 + xi3),            #    y1[k] = xi1[k+1] + xi3[k]
  Eq(y2, xi2 - xi3)               #    y2[k] = xi2[k] - xi3[k]
]

# Solve for state (xi's) and output (y's) in terms of u's
sol = solve(eqns, (xi1, xi2, xi3, y1, y2))

# Define the y and u vectors
y = Matrix([sol[y1], sol[y2]])
u = Matrix([u1, u2])

# Evaluate the Jacobian dy/du to obtain the transfer function
H = simplify(y.jacobian(u))
```

Eq. (11) shows that state-space systems have two components:

1. The map from initial state to output, $\hat{O}$.
2. The transfer function, $\hat{H}$.

We argue that the transfer function $\hat{H}$ alone is a sufficiently rich representation for the purpose of evaluating the equivalence of state-space systems. Roughly, when two

state space systems have the same transfer function, we can find initial conditions that cause the systems to have *identical input-output maps*. In other words, from the perspective of the oracle, the algorithms are indistinguishable. We state this result as Proposition 1 below.

Proposition 1. *Suppose system $i \in \{1, 2\}$ has state-space realization (A_i, B_i, C_i, D_i), initial state x_i^0, and associated transfer function $\hat{H}_i$. The following are equivalent.*

1. $\hat{H}_1 = \hat{H}_2$.
2. *There exist x_1^0 and x_2^0 such that both systems have the same input-output map.*

Under additional mild assumptions about the state-space realization, we can strengthen the forward implication of Proposition 1 to include any initial condition.

Proposition 2. *Consider the setting of Proposition 1 and further assume that both systems have* minimal *state-space realizations. The following are equivalent.*

1. $\hat{H}_1 = \hat{H}_2$.
2. *For every initialization of one system, there exists a unique initialization of the other system such that both systems have the same input-output map.*

Minimality (see Appendix A.1) is a mild assumption because from any non-minimal realization, one can construct a minimal realization with the same transfer function. For proofs of Propositions 1 and 2 and further details on how to construct minimal realizations, see Appendices A.2 and A.3, respectively.

From transfer functions to algorithms

In this paper, we study the equivalence of algorithms by analyzing their transfer functions. We always start with update equations, which lead to state-space realizations, which lead to transfer functions. For completeness, we also provide a complete characterization of when the process can be reversed, including a method to construct the update equations from the transfer function when possible, in Appendix A.3.

5.1 Algorithm equivalence

In the following three sections, we propose three notions of algorithm equivalence, each increasingly more general.

1. *Oracle equivalence.* For use when comparing algorithms that each use the same single oracle.
2. *Shift equivalence.* For use when comparing algorithms that each use the same set of oracles. Oracle equivalence is a special case of shift equivalence.
3. *LFT equivalence.* For use when comparing algorithms that use oracles that are related via a linear fractional transforms (LFTs), which we define in Section 8. Oracle equivalence and shift equivalence are both special cases of LFT equivalence.

Although the above notions of equivalence cover all examples of algorithm equivalence we have observed in practice, there are limitations to our definitions. We discuss limitations and possible extensions in Section 9.

6 Oracle equivalence

The idea behind oracle equivalence is to ask the question: "are the algorithms indistinguishable from the point of view of the oracle?" In other words, if the algorithms are suitably initialized, would using the same algorithm inputs (oracle outputs) $(u^0, u^1, \dots)$ for both algorithms always produce the same algorithm outputs (oracle inputs) $(y^0, y^1, \dots)$? If the answer is "yes", then the algorithms are *oracle equivalent*.

Motivated by Propositions 1 and 2, we will formally define oracle equivalence using the notion of transfer functions.

Definition 2 (oracle equivalence)**.** Two algorithms that use the same oracles are *oracle equivalent* if they have have the same transfer function.

Oracle equivalence is a useful notion of algorithm equivalence:

1. There are many ways to re-parameterize an algorithm that change the state-space matrices (A, B, C, D) or the internal state x^k. A sensible notion of equivalence should be independent of such transformations. Oracle equivalence achieves this independence by bypassing the state entirely and treating the algorithm as the map $(u^0, u^1, \dots) \mapsto (y^0, y^1, \dots)$.
2. Since oracle-equivalent algorithms have identical input and output sequences, many analytical properties of interest, particularly those pertaining to algorithm convergence or robustness, are preserved. For example, suppose the target problem is to minimize $f(y)$ with $y \in \mathbb{R}^n$, with solution $y^\star$ and corresponding objective value $f(y^\star)$. Further suppose f is convex and differentiable with oracle ∇f. If two algorithms are oracle-equivalent, the sequence of gradients $\|\nabla f(y)\|$, distance to the solution $\|y - y^\star\|$, and objective function values $f(y) - f(y^\star)$ evolve identically, so they have the same worst-case convergence rate. Moreover, even if the oracle is noisy (e.g., suffers from additive or multiplicative noise, or even adversarial noise), from the point of view of the oracle, the algorithms are indistinguishable and any analytical property that involves only the oracle sequence will be the same.

Invariance under linear state transformations

Oracle equivalence (Definition 2) is invariant under linear transformations of state. Specifically, define $\tilde{x}^k = Tx^k$ for each k, where T is an invertible matrix. The state-space equations (9) expressed in terms of the new state variable $\tilde{x}^k$ become

$$\tilde{x}^{k+1} = TAT^{-1}\tilde{x}^k + TBu^k, \tag{14a}$$

$$y^k = CT^{-1}\tilde{x}^k + Du^k. \tag{14b}$$

Substituting into Eq. (12), we can verify that both systems have the same transfer function.[4] If we initialize the transformed system with $\tilde{x}^0 = Tx^0$ and apply the same input $(u_0, u_1, \dots)$ to both systems, we will obtain the same output $(y_0, y_1, \dots)$, although the respective states x^k and $\tilde{x}^k$ will generally be different. This invariance is the key to understanding when two optimization algorithms are the same, even

[4] Both systems will also have the same *Markov parameters* and *Hankel matrices* (see Appendix A.1).

if they look different as written. For example, this idea alone suffices to show that Algorithms 3.1 and 3.2 are equivalent.

When using linear state transformations, the number of states (size of the A matrix) is preserved. However, realizations with a different number of states can also be oracle equivalent. Although a realization with a larger A matrix will generally lead to a transfer function with higher degree (via Eq. (12)), there may be common factors that cancel from the numerator and denominator, leading to lower-degree transfer functions that could have been obtained from a realization with a smaller A matrix. This idea is related to the notion of minimality (see Appendix A.1) and explains why Algorithms 3.3 and 3.4 are equivalent.

6.1 Examples of oracle equivalence

Now, we will revisit the first and second motivating examples and apply Definition 2 to show oracle equivalence. Specifically, we will compute transfer functions using Eq. (12) and verify that they are the same.

Algorithms 3.1 and 3.2

The state-space realization and transfer function of Algorithms 3.1 and 3.2 are

$$\hat{H}_{3.1} = \left[\begin{array}{cc|c} 2 & -1 & -\frac{1}{10} \\ 1 & 0 & 0 \\ \hline 2 & -1 & 0 \end{array}\right] = \begin{bmatrix} 2 & -1 \end{bmatrix} \left(zI - \begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix} \right)^{-1} \begin{bmatrix} -\frac{1}{10} \\ 0 \end{bmatrix} = \frac{-2z+1}{10(z-1)^2},$$

$$\hat{H}_{3.2} = \left[\begin{array}{cc|c} 1 & -1 & -\frac{1}{5} \\ 0 & 1 & \frac{1}{10} \\ \hline 1 & 0 & 0 \end{array}\right] = \begin{bmatrix} 1 & 0 \end{bmatrix} \left(zI - \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} -\frac{1}{5} \\ \frac{1}{10} \end{bmatrix} = \frac{-2z+1}{10(z-1)^2}.$$

Since $\hat{H}_{3.1} = \hat{H}_{3.2}$, Algorithms 3.1 and 3.2 are oracle-equivalent by Definition 2. Algorithm 3.1 can also be transformed to Algorithm 3.2 as in Eq. (14) via $T = \left[\begin{smallmatrix} 2 & -1 \\ -1 & 1 \end{smallmatrix}\right]$.

Algorithms 3.3 and 3.4

The state-space realization and transfer function of Algorithms 3.3 and 3.4 are

$$\hat{H}_{3.3} = \left[\begin{array}{cc|c} 3 & -2 & \frac{1}{5} \\ 1 & 0 & 0 \\ \hline -1 & 2 & 0 \end{array}\right] = \begin{bmatrix} -1 & 2 \end{bmatrix} \left(zI - \begin{bmatrix} 3 & -2 \\ 1 & 0 \end{bmatrix} \right)^{-1} \begin{bmatrix} \frac{1}{5} \\ 0 \end{bmatrix} = \frac{-z+2}{5(z^2-3z+2)} = \frac{-1}{5(z-1)},$$

$$\hat{H}_{3.4} = \left[\begin{array}{c|c} 1 & -\frac{1}{5} \\ \hline 1 & 0 \end{array}\right] = \begin{bmatrix} 1 \end{bmatrix} \left(zI - \begin{bmatrix} 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} -\frac{1}{5} \end{bmatrix} = \frac{-1}{5(z-1)}.$$

Since $\hat{H}_{3.3} = \hat{H}_{3.4}$, Algorithms 3.3 and 3.4 are oracle-equivalent by Definition 2. The transfer functions are the same due to the cancellation of the common factor $(z-2)$ in the numerator and denominator of $\hat{H}_{3.3}$.

Algorithms 1.1–1.4

Using the same approach as above, we can derive the transfer functions for Algorithms 1.1–1.4 and show that they are all equal to $\hat{H}(z) = -\frac{\eta(2z-1)}{z(z-1)}$. Therefore, Algorithms 1.1–1.4 are oracle-equivalent by Definition 2.

Accelerated gradient methods

Accelerated gradient methods are a class of optimization algorithms designed to improve the convergence speed of gradient-based methods, especially for convex optimization problems. Accelerated methods incorporate momentum-like terms and interpolated iterates that help the optimization process converge faster. Two well-know methods include Polyak's Heavy Ball (HB) [30] and Nesterov's Accelerated Gradient Method (NAG) [31], shown below as Algorithms 6.1 and 6.2. These techniques and their stochastic variants are widely used in machine learning, signal processing, and numerical optimization.

Algo. 6.1 (Polyak's Heavy Ball)

for $k = 0, 1, 2, \ldots$ **do**
$x^{k+1} = x^k - \alpha\nabla f(x^k) + \beta(x^k - x^{k-1})$
end for

Algo. 6.2 (Nesterov's Method)

for $k = 0, 1, 2, \ldots$ **do**
$y^k = x^k + \beta(x^k - x^{k-1})$
$x^{k+1} = y^k - \alpha\nabla f(y^k)$
end for

Several works have proposed *unified* momentum algorithms and associated analyses that generalize HB and NAG and allow the algorithm designer to interpolate between both algorithms. Examples include: Triple Momentum Method [32], Quasi-Hyperbolic Momentum [33], Stochastic Unified Method [34], and Unified Stochastic Momentum [35], listed below as Algorithms 6.3–6.6, respectively.

Algo. 6.3 (Triple Momentum Method)

for $k = 0, 1, 2, \ldots$ **do**
$y^k = x^k + \gamma(x^k - x^{k-1})$
$x^{k+1} = x^k - \alpha\nabla f(y^k) + \beta(x^k - x^{k-1})$
end for

Algo. 6.4 (Quasi-Hyperbolic Momentum)

for $k = 0, 1, 2, \ldots$ **do**
$g^{k+1} = \beta g^k + (1-\beta)\nabla f(\theta^k)$
$\theta^{k+1} = \theta^k - \alpha((1-\nu)\nabla f(\theta^k) + \nu g^{k+1})$
end for

Algo. 6.5 (Stochastic Unified Momentum)

for $k = 0, 1, 2, \ldots$ **do**
$y^{k+1} = x^k - \alpha\nabla f(x^k)$
$q^{k+1} = x^k - s\alpha\nabla f(x^k)$
$x^{k+1} = y^{k+1} + \beta(q^{k+1} - q^k)$
end for

Algo. 6.6 (Unified Stochastic Momentum)

for $k = 0, 1, 2, \ldots$ **do**
$m^k = \mu m^{k-1} - \eta\nabla f(x^k)$
$x^{k+1} = x^k + m^k + \lambda\mu(m^k - m^{k-1})$
end for

The transfer functions for HB and NAG are clearly different:

$$\hat{H}_{6.1} = \frac{-\alpha z}{(z-1)(z-\beta)}, \qquad \hat{H}_{6.2} = \frac{-\alpha(1+\beta)(z - \frac{\beta}{1+\beta})}{(z-1)(z-\beta)}.$$

However, the transfer functions for Algorithms 6.3–6.6 are:

$$\hat{H}_{6.3}(z) = -\frac{\alpha(1+\gamma)(z-\frac{\gamma}{1+\gamma})}{(z-1)(z-\beta)}, \qquad \hat{H}_{6.4}(z) = -\frac{\alpha(1-\beta\nu)(z-\frac{\beta(1-\nu)}{1-\beta\nu})}{(z-1)(z-\beta)},$$
$$\hat{H}_{6.5}(z) = -\frac{\alpha(1+\beta s)(z-\frac{\beta s}{1+\beta s})}{(z-1)(z-\beta)}, \qquad \hat{H}_{6.6}(z) = -\frac{\eta(1+\lambda\mu)(z-\frac{\lambda\mu}{1+\lambda\mu})}{(z-1)(z-\mu)}.$$

Each of the above transfer functions are of the form $-\frac{a(z-c)}{(z-1)(z-b)}$ for some a, b, c and can be made equal to one another (oracle equivalent) via suitable choices of the algorithm parameters. In other words, these algorithms parameterize the same space of possible algorithms (which also includes HB and NAG); they are equally general.

Distributed optimization

For our final example of this section, we consider *synchronous distributed optimization*, where a network of computing nodes work collaboratively to solve the problem

$$\underset{x\in\mathbb{R}^d}{\text{minimize}} \quad \sum_{i=1}^{n} f_i(x). \tag{15}$$

Each node i maintains a local state $x_i^k \in \mathbb{R}^d$ and can access the oracle ∇f_i. At every timestep, each node can *gossip* (obtain the local states of its neighboring nodes x_j^k), evaluate its local oracle, and perform computations to update its local state. There are two goals: (1) *consensus*: the nodes' local states should converge to a common value, and (2) *optimality*: the common value should be $x^\star$, a solution of Eq. (15). For convenience, we use the shorthand notation

$$x^k := \begin{bmatrix} x_1^k \\ \vdots \\ x_n^k \end{bmatrix} \quad \text{and} \quad \nabla f(x^k) := \begin{bmatrix} \nabla f_1(x_1^k) \\ \vdots \\ \nabla f_n(x_n^k) \end{bmatrix}.$$

Gossip is modeled as matrix multiplication Wx^k, where $W = \tilde{W} \otimes I_d$ and $\tilde{W} \in \mathbb{R}^{n\times n}$ is a (typically sparse) row-stochastic matrix; it satisfies $0 \leq \tilde{W} \leq 1$ and $W\mathbf{1} = \mathbf{1}$.

Under suitable assumptions on W, the algorithm $x^{k+1} = Wx^k$ achieves consensus at a linear rate, but the consensus value will be the mean of the x_i^0 rather than $x^\star$ (no optimality). Likewise, if the f_i are smooth and strongly convex, gradient descent $x^{k+1} = x^k - \alpha\nabla f(x^k)$ achieves local but not global optimality: each node converges to the minimizer of its local f_i rather than the minimizer of $\sum_{i=1}^n f_i$. A simple algorithm that achieves both consensus and optimality is distributed gradient descent [36], which combines features of both gossip and gradient descent: $x^{k+1} = Wx^k - \alpha_k\nabla f(x^k)$. However, this algorithm only converges sublinearly, even with strongly convex f_i, and requires a diminishing stepsize $\alpha_k \to 0$ to converge at all.

The first algorithm to guarantee linear convergence for strongly convex f_i was EXTRA [37], and since then many papers have developed new algorithms or refined

existing ones to solve Eq. (15) with a linear convergence rate. Two such well-known algorithms are NIDS [38] and Exact Diffusion [39], shown below.

Algo. 6.7 NIDS

for $k = 0, 1, 2, \dots$ **do**
$x^{k+2} = W\big(2x^{k+1} - x^k - \alpha\nabla f(x^{k+1}) + \alpha\nabla f(x^k)\big)$
end for

Algo. 6.8 Exact Diffusion

for $k = 0, 1, 2, \dots$ **do**
$x^{k+1} = w^k - \alpha\nabla f(w^k)$
$y^k = x^{k+1} + w^k - x^k$
$w^{k+1} = Wy^k$
end for

These algorithms were developed using different approaches. NIDS used a *gradient-differencing* intuition similar to EXTRA to achieve linear convergence (storing the past gradient and updating based on the difference, as in Algorithm 6.7). In contrast, Exact Diffusion used an *adapt-correct-combine* concept (corresponding to the three update equations in Algorithm 6.8, respectively).

However, NIDS and Exact Diffusion are (oracle) equivalent [40]! We can detect this equivalence automatically by computing the transfer function for each algorithm. In this case, we obtain $\hat{H}_{6.7}(z) = \hat{H}_{6.8}(z) = -\alpha(z-1)W(z^2I - 2zW + W)^{-1}$.

7 Shift equivalence

Now consider Algorithms 3.5 and 3.6 from the third motivating example. We can calculate that the algorithms have different transfer functions:

$$\hat{H}_{3.5} = \begin{bmatrix} \frac{-1}{z-1} & \frac{1}{z-1} \\ \frac{2z-1}{z-1} & \frac{-1}{z-1} \end{bmatrix}, \qquad \hat{H}_{3.6} = \begin{bmatrix} \frac{-1}{z-1} & \frac{z}{z-1} \\ \frac{2z-1}{z(z-1)} & \frac{-1}{z-1} \end{bmatrix}, \tag{16}$$

so they are not oracle-equivalent. We can represent the equations for Algorithms 3.5 and 3.6 using block diagrams that are *unrolled in time*; see Fig. 3. Based on the diagram, it is clear that the algorithms are just shifted versions of one another. If we initialize Algorithm 3.6 using $\xi_2^0 = x_1^1$ and $\xi_3^0 = x_3^0 - x_1^1$, then it will make the same oracle calls as Algorithm 3.5, but with a time shift.

This example motivates us to define *shift equivalence*. As with oracle equivalence, we ask whether the algorithms are indistinguishable from the point of view of the oracle for suitably chosen input channel delays and state initializations. Before we define shift equivalence, we will formalize the notion of shifting.

Shifting (delaying) a semi-infinite sequence $(y^0, y^1, \dots)$ by m time steps corresponds to multiplication of its z-transform by z^{-m}:

$$\mathcal{Z}\Big[(y^0, y^1, y^2, \dots)\Big] = \sum_{k=0}^{\infty} y^k z^{-k} = \hat{y}(z), \quad \text{and}$$

$$\mathcal{Z}\Big[(\underbrace{0, 0, \dots, 0}_{m \text{ times}}, y^0, y^1, y^2, \dots)\Big] = \sum_{k=0}^{\infty} y^k z^{-m-k} = z^{-m}\hat{y}(z).$$

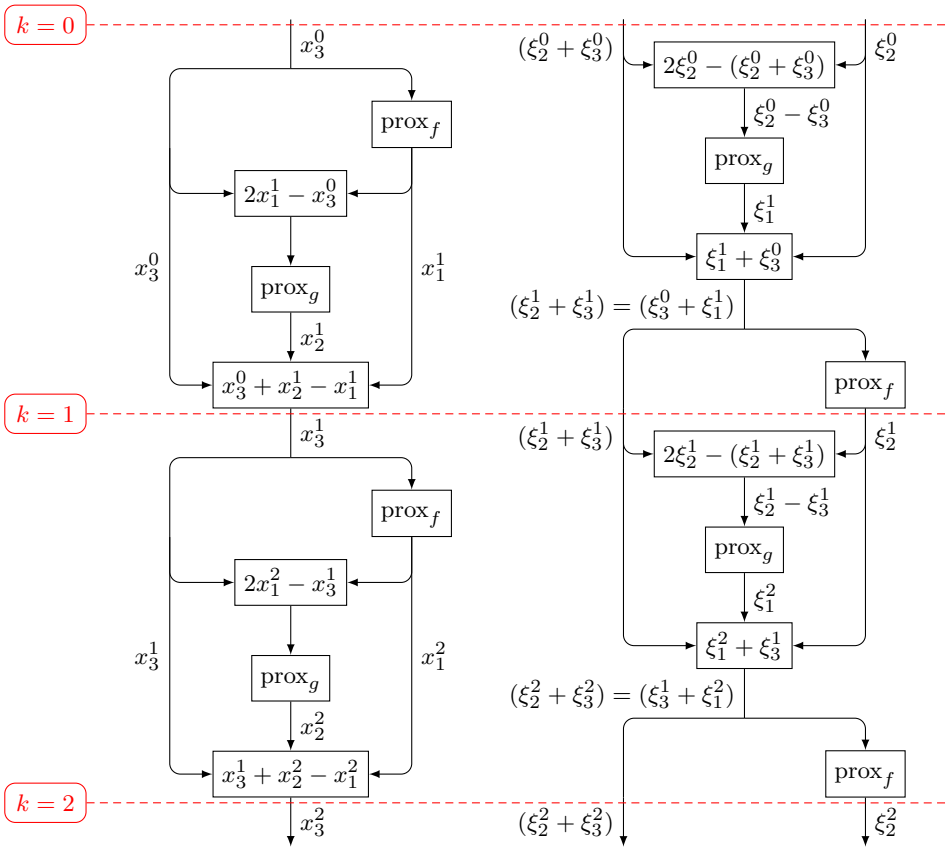

Fig. 3: Block diagrams representing Algorithm 3.5 (left) and Algorithm 3.6 (right). These algorithms are *shift equivalent* because when suitably initialized, they make the same calls to the oracles, albeit with a time shift. The updates are exactly the same for both algorithms, but using transformed variables.

Since the oracle ϕ applies element-wise to each y^k and $\phi(0) = 0$ by assumption, the oracle ϕ commutes with the shift operation. We can represent this relationship by a commutative diagram; see Fig. 4.

For a vector-valued signal, we can delay each component by a different amount. This motivates the definition of the *multi-shift*.

Definition 3 (multi-shift)**.** We define the *multi-shift* $\hat{\Delta}_m(z)$ for nonnegative integers $m := (m_1, \ldots, m_p)$ as the transfer function

$$\hat{\Delta}_m(z) := \begin{bmatrix} z^{-m_1} & & 0 \\ & \ddots & \\ 0 & & z^{-m_p} \end{bmatrix}$$

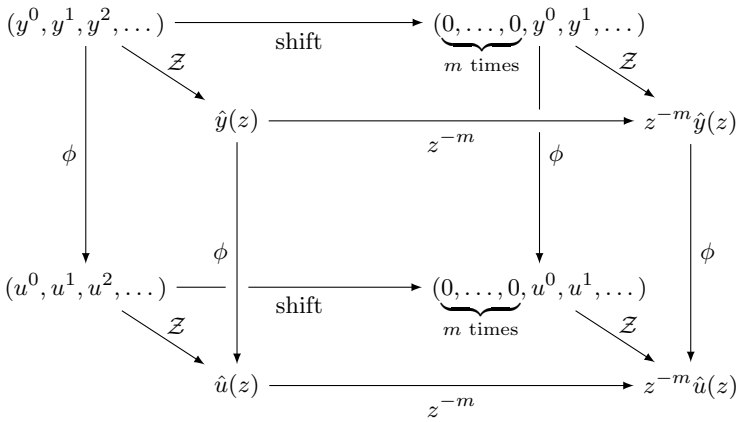

Fig. 4: Commutative diagram visualizing that the shift (delay) operation commutes with the application of the oracle ϕ. The foreground shows the z-transformed versions of the signals, where the shift becomes multiplication by a power of z^{-1}.

For example, consider the semi-infinite sequence $(y^0, y^1, y^2, \dots)$, where each y^k is partitioned into blocks y_1^k, y_2^k, y_3^k, where the y_i^k are the same size for all k. Then,

$$\hat{\Delta}_{(1,0,2)}(z)\hat{y}(z) = \mathcal{Z}\left[\left(\begin{bmatrix}0\\ y_2^0\\ 0\end{bmatrix}, \begin{bmatrix}y_1^0\\ y_2^1\\ 0\end{bmatrix}, \begin{bmatrix}y_1^1\\ y_2^2\\ y_3^0\end{bmatrix}, \begin{bmatrix}y_1^2\\ y_2^3\\ y_3^1\end{bmatrix}, \dots, \begin{bmatrix}y_1^{k-1}\\ y_2^k\\ y_3^{k-2}\end{bmatrix}, \dots\right)\right].$$

The multi-shift also commutes with any time-invariant oracle $\Phi = (\phi_1, \dots, \phi_p)$. In terms of z-transforms, $\Phi(\hat{y}) = \hat{\Delta}_m^{-1}\Phi(\hat{\Delta}_m\hat{y})$. By rearranging the block diagram, we can move the multi-shifts from the oracle to the algorithm; see Fig. 5.

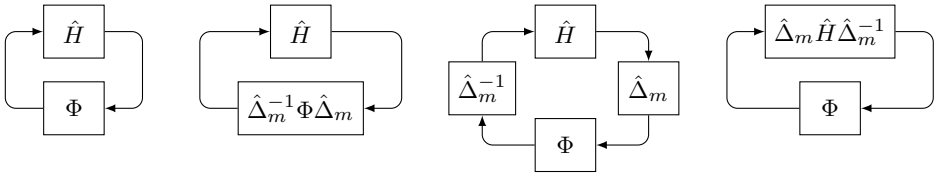

Fig. 5: Equivalent block diagram representing shift equivalence. We use the fact that the oracle Φ commutes with any multi-shift $\hat{\Delta}_m$. However, $\hat{\Delta}_m$ need not commute with $\hat{H}$, which means equivalent algorithms can have different transfer functions.

The transformation in Fig. 5 shows that from the point of view of the oracle Φ, the algorithms $\hat{H}$ and $\hat{\Delta}_m\hat{H}\hat{\Delta}_m^{-1}$ are indistinguishable. This motivates our definition of *shift equivalence*.

Definition 4 (shift equivalence)**.** Suppose we are given two LTI algorithms that each use the same p oracles and have transfer functions $\hat{H}_1$ and $\hat{H}_2$, respectively. We say

they are *shift-equivalent* and write $\hat{H}_1 \sim \hat{H}_2$ if there exists a multi-shift $\hat{\Delta}_m$ such that

$$\hat{H}_1 = \hat{\Delta}_m \hat{H}_2 \hat{\Delta}_m^{-1}.$$

Our choice of the word *equivalence* is justified by the fact that shift equivalence is an *equivalence relation*.

Lemma 3. *Shift equivalence, as defined in Definition 4, is an* equivalence relation. *That is, it satisfies the properties of reflexivity, symmetry, and transitivity.*

Proof. See Appendix B.1. □

Remark 2. Oracle equivalence is a special case of shift equivalence (with $\hat{\Delta}_m = I$). Moreover, shift equivalence reduces to oracle equivalence when $p = 1$ (a single oracle). In this case, the transfer functions and multi-shifts are scalars rather than matrices, so they trivially commute: $z^{-m}\hat{H}_1 = \hat{H}_2 z^{-m} \iff \hat{H}_1 = \hat{H}_2$.

Efficient enumeration of shift-equivalent algorithms

Given an algorithm $\hat{H}$ using oracles $\Phi = (\phi_1, \ldots, \phi_p)$, how can we generate all possible shift-equivalent algorithms? In other words, what are the possible transfer functions $\hat{H}'$ and multi-shifts $\hat{\Delta}_m$ such that

$$\hat{H}'(z) = \begin{bmatrix} z^{-m_1} & & 0 \\ & \ddots & \\ 0 & & z^{-m_p} \end{bmatrix} \hat{H} \begin{bmatrix} z^{m_1} & & 0 \\ & \ddots & \\ 0 & & z^{m_p} \end{bmatrix}.$$

Since $\hat{H}'$ must be a proper transfer function (or strictly proper, depending on whether we require explicit implementations; refer to Appendix A.3), each $\hat{H}_{ij}(z)z^{m_j - m_i}$ must be proper. To determine the set of possible $\hat{H}'$, let r_{ij} be the *relative degree* of $\hat{H}_{ij}$. That is, write $\hat{H}_{ij}(z) = N_{ij}(z)/D_{ij}(z)$ (ratio of polynomials), and

$$r_{ij} := \begin{cases} +\infty & \text{if } \hat{H}_{ij}(z) = 0 \\ \deg(D_{ij}) - \deg(N_{ij}) & \text{otherwise} \end{cases}$$

Then, properness of $\hat{H}'$ amounts to finding m_i such that

$$-r_{ij} \leq m_i - m_j \leq r_{ji} \qquad \text{for all } i \neq j. \tag{17}$$

Since the set of feasible m_i is translation-invariant, we can normalize each solution so that $\min_i m_i = 0$, and each distinct solution $\{m_i\}$ will correspond to a distinct $\hat{H}'$ that is shift-equivalent to $\hat{H}$. For an example of how we can enumerate solutions, see the primal-dual three-operator splitting example in Section 7.1.

Efficient determination of shift equivalence

Given two algorithms $\hat{H}$ and $\hat{H}'$ that use the same oracles $(\phi_1, \dots, \phi_p)$, we can efficiently check whether these algorithms are shift-equivalent by carrying out the following steps.

1. Check to make sure $\hat{H}_{ii} = H'_{ii}$ for all i (the diagonal entries must always match). Otherwise, they are not equivalent.
2. Check to make sure that for all $i \neq j$, either $\hat{H}_{ij} = \hat{H}'_{ij} = 0$, or $H_{ij} \neq 0$ and $\hat{H}'_{ij} \neq 0$. In other words, $\hat{H}$ and $\hat{H}'$ must have matching sparsity patterns. Otherwise, they are not equivalent.
3. For all $i \neq j$ such that $\hat{H}_{ij} \neq 0$, check to make sure that $H'_{ij}(z)/\hat{H}_{ij}(z) = z^{b_{ij}}$ for some integers b_{ij}. In other words, corresponding entries of the algorithm's transfer functions must by related by multiplication by a power of z.
4. Consider the set of linear equations $b_{ij} = m_j - m_i$ for all $i \neq j$ such that $\hat{H}_{ij} \neq 0$. Write the corresponding linear equations compactly as $A^\top m = b$. If this system of equations has a solution, then $\hat{H} \sim \hat{H}'$. Otherwise, they are not equivalent. Note that if a solution exists, we can always find a solution with integer m, since A is an incidence matrix and therefore is totally unimodular.

7.1 Examples of shift equivalence

Douglas–Rachford splitting and ADMM

As computed in Eq. (16), the transfer functions for Algorithms 3.5 and 3.6 are given by $\hat{H}_{3.5} = \begin{bmatrix} \frac{-1}{z-1} & \frac{1}{z-1} \\ \frac{2z-1}{z-1} & \frac{-1}{z-1} \end{bmatrix}$ and $\hat{H}_{3.6} = \begin{bmatrix} \frac{-1}{z-1} & \frac{z}{z-1} \\ \frac{2z-1}{z(z-1)} & \frac{-1}{z-1} \end{bmatrix}$. We see that they are related via $\hat{H}_{3.5} = \begin{bmatrix} z^{-1} & 0 \\ 0 & 1 \end{bmatrix} \hat{H}_{3.6} \begin{bmatrix} z & 0 \\ 0 & 1 \end{bmatrix}$. Therefore, $\hat{H}_{3.5} \sim \hat{H}_{3.6}$.

Primal-dual three-operator splitting

For our next example, we consider algorithms for solving the optimization problem

$$\text{minimize } f(x) + g(Ax) + h(x)$$

using the oracles $\text{prox}_{\tau f}$, $\text{prox}_{\sigma g^*}$, and ∇h. Only recently have methods been proposed to solve this problem. Examples include the Condat–Vũ algorithm independently proposed by Condat and Vũ [41, 42], the primal-dual three-operator (PD3O) algorithm [43], and the primal-dual Davis–Yin (PDDY) algorithm [44]. See [45] and references therein for a recent survey on this problem. To illustrate our approach, we will focus on the primal-dual three-operator algorithm (PD3O) [43], shown below.

Algo. 7.1 Primal-dual three-operator splitting (PD3O)

for $k = 0, 1, 2, \ldots$ **do**
$x^k = \mathrm{prox}_{\tau f}(z^k)$
$s^{k+1} = \mathrm{prox}_{\sigma g^*}\Big((I - \tau\sigma AA^\top)s^k + \sigma A\big(2x^k - z^k - \tau\nabla h(x^k)\big)\Big)$
$z^{k+1} = x^k - \tau\nabla h(x^k) - \tau A^\top s^{k+1}$
end for

The state-space realization and transfer function for PD3O is[5]

$$\hat{H}_{7.1}(z) = \left[\begin{array}{cc|ccc} 0 & 0 & 0 & I & 0 \\ 0 & 0 & I & -\tau A^\top & -\tau I \\ \hline 0 & I & 0 & 0 & 0 \\ I - \tau\sigma AA^\top & -\sigma A & 2\sigma A & 0 & -\tau\sigma A \\ 0 & 0 & I & 0 & 0 \end{array}\right] = \left[\begin{array}{ccc} \frac{1}{z} & \frac{-\tau A^\top}{z} & \frac{-\tau}{z} \\ \frac{\sigma(2z-1)A}{z} & \frac{1}{z} & \frac{-\sigma\tau(z-1)A}{z} \\ 1 & 0 & 0 \end{array}\right]$$

In [43], the authors show a reformulation of PD3O and state that it was obtained by changing the order of the variables and substituting $\bar{x}^k = 2x^k - z^k - \tau\nabla h(x^k) - \tau A^\top s^k$. After these changes, the reformulation is given by Algorithm 7.2 below.

Algo. 7.2 Reformulation of PD3O

for $k = 0, 1, 2, \ldots$ **do**
$s^{k+1} = \mathrm{prox}_{\sigma g^*}(s^k + \sigma A\bar{x}^k)$
$x^{k+1} = \mathrm{prox}_{\tau f}(x^k - \tau\nabla h(x^k) - \tau A^\top s^{k+1})$
$\bar{x}^{k+1} = 2x^{k+1} - x^k + \tau\nabla h(x^k) - \tau\nabla h(x^{k+1})$
end for

$$\hat{H}_{7.2} = \left[\begin{array}{ccc} \frac{1}{z} & -\tau A^\top & \frac{-\tau}{z} \\ \frac{\sigma(2z-1)A}{z^2} & \frac{1}{z} & \frac{-\sigma\tau(z-1)A}{z^2} \\ 1 & 0 & 0 \end{array}\right]$$

We can obtain the equivalence between Algorithms 7.1 and 7.2 automatically. Indeed, Algorithms 7.1 and 7.2 are shift equivalent because

$$\hat{H}_{7.2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & z^{-1} & 0 \\ 0 & 0 & 1 \end{bmatrix} \hat{H}_{7.1} \begin{bmatrix} 1 & 0 & 0 \\ 0 & z & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

This is not the only possible shift-equivalence transformation. Applying the method outlined in Eq. (17), the relative degree matrix of $\hat{H}_{7.1}$ is given by

$$[r_{ij}] = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & \infty & \infty \end{bmatrix}.$$

We seek nonnegative integers (m_1, m_2, m_3) normalized so that $\min_i m_i = 0$ satisfying

$$0 \le m_2 - m_1 \le 1, \qquad 0 \le m_3 - m_1 \le 1, \qquad 0 \le m_2 - m_3 \le \infty.$$

[5] We have removed identity matrices from the transfer function to simplify exposition. For example, entries in $\hat{H}_{7.1}(z)$ that read $\frac{1}{z}$ should be replaced by $\frac{1}{z}I$.

Algorithm 7.1 corresponds to the trivial solution $(0,0,0)$, and Algorithm 7.2 corresponds to $(0,1,0)$. By inspection, we can find a third solution, $(0,1,1)$. This solution corresponds to the new algorithm

$$\hat{H}_{7.3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & z^{-1} & 0 \\ 0 & 0 & z^{-1} \end{bmatrix} \hat{H}_{7.1} \begin{bmatrix} 1 & 0 & 0 \\ 0 & z & 0 \\ 0 & 0 & z \end{bmatrix}.$$

One possible realization of Algorithm 7.3 is given below.

Algo. 7.3 Another reformulation of PD3O

for $k = 0, 1, 2, \dots$ **do**
$y^k = \operatorname{prox}_{\sigma g^*}\big(\sigma A(2w^k - \tau\nabla h(w^k)) - x^k\big)$
$x^{k+1} = \sigma A\big(w^k - \tau\nabla h(w^k)\big) - y^k$
$w^{k+1} = \operatorname{prox}_{\tau f}(x^{k+1})$
end for

$$\hat{H}_{7.3} = \begin{bmatrix} \frac{1}{z} & -\tau A^\top & -\tau \\ \frac{\sigma(2z-1)A}{z^2} & \frac{1}{z} & \frac{-\sigma\tau(z-1)A}{z} \\ \frac{1}{z} & 0 & 0 \end{bmatrix}$$

We stress that these equivalences are tedious to work out by hand, and since there are now three oracles, the equivalences are far from obvious. For example, here is another way to realize Algorithm 7.2. This time, the transfer functions are the same, so Algorithms 7.2 and 7.4 are oracle-equivalent.

Algo. 7.4 Yet another reformulation of PD3O

for $k = 0, 1, 2, \dots$ **do**
$y^k = \operatorname{prox}_{\sigma g^*}(2\sigma A w^k - x^k)$
$z^k = \operatorname{prox}_{\tau f}(w^k - \tau A^\top y^k)$
$x^{k+1} = \sigma A\big(w^k - \tau\nabla h(z^k)\big) - y^k$
$w^{k+1} = z^k - \tau\nabla h(z^k)$
end for

$$\hat{H}_{7.4} = \begin{bmatrix} \frac{1}{z} & -\tau A^\top & \frac{-\tau}{z} \\ \frac{\sigma(2z-1)A}{z^2} & \frac{1}{z} & \frac{-\sigma\tau(z-1)A}{z^2} \\ 1 & 0 & 0 \end{bmatrix}$$

8 LFT equivalence

In Sections 6 and 7, we considered equivalence between algorithms that use the same oracles. In this section, we consider equivalence between algorithms that use *different but related* oracles.

In convex optimization, algorithm conjugation naturally relates some oracles to others [46][13, §2]: for example, if $(\partial f)(x) := \{g \mid f(y) \geq f(x) + g^\top(y-x) \text{ for all } y\}$ is the subdifferential of f, $\operatorname{prox}_f(v) := \operatorname{argmin}_x\big(f(x) + \frac{1}{2}\|x-v\|^2\big)$ is the proximal operator of f, and $f^*(y) := \sup_x\{x^\top y - f(x)\}$ is the Fenchel conjugate of f [25, §3], we have the following identities relating the different operators.

- $y \in \partial f(x) \iff x \in \partial f^*(y)$,
- $y = \operatorname{prox}_{tf}(x) \iff x \in y + t\partial f(y)$
- $x = \operatorname{prox}_{tf}(x) + t\operatorname{prox}_{\frac{1}{t}f^*}(\frac{1}{t}x)$ (Moreau's identity)

We can rewrite any algorithm in terms of different, also easily computable, oracles using these identities. Consider a simple example: we will obfuscate the proximal gradient method (Algorithm 8.1 [24, §10][47]) by rewriting it in terms of the conjugate of the original oracle prox_g, using Moreau's identity, as Algorithm 8.2 [48]. These are the same as our motivating examples of Algorithms 3.7 and 3.8.

Algo. 8.1 Proximal gradient

for $k = 0, 1, 2, \ldots$ **do**
$y^k = x^k - t\nabla f(x^k)$
$x^{k+1} = \text{prox}_{tg}(y^k)$
end for

Algo. 8.2 Conjugate proximal gradient

for $k = 0, 1, 2, \ldots$ **do**
$y^k = x^k - t\nabla f(x^k)$
$x^{k+1} = y^k - t\text{prox}_{\frac{1}{t}g^*}(\frac{1}{t}y^k)$
end for

We can also use the relationship between the proximal and subdifferential operators to obtain versions of Algorithms 8.1 and 8.2 that use subdifferentials instead.

Algo. 8.3 Subdifferential

for $k = 0, 1, 2, \ldots$ **do**
$y^k = x^k - t\nabla f(x^k)$
$x^{k+1} \in y^k - t\partial g(x^{k+1})$
end for

Algo. 8.4 Conjugate Subdifferential

for $k = 0, 1, 2, \ldots$ **do**
$y^k = x^k - t\nabla f(x^k)$
$x^{k+1} \in \partial g^*\left(\frac{1}{t}(y^k - x^{k+1})\right)$
end for

Note that the update equations for Algorithms 8.3 and 8.4 involving subdifferentials are implicit. The transfer functions for Algorithms 8.1–8.4 are shown below, along with their associated oracles. We use the symbol $\circ$ to show that an algorithm is used with a given set of oracles.

$$\begin{aligned}
&\text{Algo. 8.1}: \begin{bmatrix} 0 & \frac{1}{z} \\ -t & \frac{1}{z} \end{bmatrix} \circ (\nabla f, \text{prox}_{tg}), \quad \text{Algo. 8.2}: \begin{bmatrix} \frac{-t}{z-1} & \frac{-t}{z-1} \\ \frac{-z}{z-1} & \frac{-1}{z-1} \end{bmatrix} \circ (\nabla f, \text{prox}_{\frac{1}{t}g^*}), \\
&\text{Algo. 8.3}: \begin{bmatrix} \frac{-t}{z-1} & \frac{-t}{z-1} \\ \frac{-tz}{z-1} & \frac{-tz}{z-1} \end{bmatrix} \circ (\nabla f, \partial g), \quad \text{Algo. 8.4}: \begin{bmatrix} 0 & \frac{1}{z} \\ -1 & \frac{1-z}{tz} \end{bmatrix} \circ (\nabla f, \partial g^*).
\end{aligned} \tag{18}$$

Although the transfer functions of the algorithms change when we rewrite the algorithm to call a different oracle, the sequence of states is preserved (x^k and y^k have the same values for all algorithms provided they are initialized the same way). This motivates us to define a general notion of equivalence that applies when two algorithms use different oracles that are related in a particular way.

Definition 5 (Operator graph)**.** Given an oracle $\Phi : \mathcal{V}^p \to \mathcal{V}^p$, we define its *graph* as the set of possible input-output pairs (in the z-domain). We adopt the linear algebraic notation $\mathcal{R}$ (range) overloaded as follows:

$$\mathcal{R}\begin{bmatrix} I \\ \Phi \end{bmatrix} := \left\{ \begin{bmatrix} \hat{y} \\ \Phi(\hat{y}) \end{bmatrix} \;\middle|\; \hat{y} = \mathcal{Z}[y^k], \text{ where } y^k \in \mathcal{V}^p \text{ for } k = 0, 1, \ldots \right\}.$$

Likewise, given an algorithm $\hat{H}$, we define its *dual graph* as:

$$\mathcal{R}\begin{bmatrix}\hat{H}\\ I\end{bmatrix} := \left\{ \begin{bmatrix}\hat{H}\hat{u}\\ \hat{u}\end{bmatrix} \;\middle|\; \hat{u} = \mathcal{Z}[u^k],\ \text{where } u^k \in \mathcal{V}^p \text{ for } k = 0, 1, \dots \right\}.$$

Definition 6 (Linearly equivalent oracles)**.** Let Φ_1 and Φ_2 be oracles. We say that Φ_1 is *linearly equivalent* to Φ_2 and we write $\Phi_1 \overset{\hat{M}}{\sim} \Phi_2$, if their graphs are related by an invertible linear transformation $\hat{M}$. In other words, $\Phi_1 \overset{\hat{M}}{\sim} \Phi_2$ if

$$\mathcal{R}\begin{bmatrix}I\\ \Phi_1\end{bmatrix} = \hat{M}\,\mathcal{R}\begin{bmatrix}I\\ \Phi_2\end{bmatrix}.$$

This is equivalent to saying that:

- If $\hat{u}_1 = \Phi_1(\hat{y}_1)$, there exists $\hat{y}_2, \hat{u}_2$ such that $\hat{u}_2 = \Phi_2(\hat{y}_2)$ and $\begin{bmatrix}\hat{y}_1\\ \hat{u}_1\end{bmatrix} = \hat{M}\begin{bmatrix}\hat{y}_2\\ \hat{u}_2\end{bmatrix}$, and
- If $\hat{u}_2 = \Phi_2(\hat{y}_2)$, there exists $\hat{y}_1, \hat{u}_1$ such that $\hat{u}_1 = \Phi_1(\hat{y}_1)$ and $\begin{bmatrix}\hat{y}_1\\ \hat{u}_1\end{bmatrix} = \hat{M}\begin{bmatrix}\hat{y}_2\\ \hat{u}_2\end{bmatrix}$.

We will omit $\hat{M}$ and simply write $\Phi_1 \sim \Phi_2$ to mean that there exists some invertible $\hat{M}$ such that $\Phi_1 \overset{\hat{M}}{\sim} \Phi_2$.

For example, $\operatorname{prox}_f \sim \partial f$ because:

$$\begin{cases} x = y + t\partial f(y) \\ y = \operatorname{prox}_{tf}(x) \end{cases} \quad \Longleftrightarrow \quad \begin{bmatrix} x \\ \operatorname{prox}_{tf}(x) \end{bmatrix} = \begin{bmatrix} 1 & t \\ 1 & 0 \end{bmatrix} \begin{bmatrix} y \\ \partial f(y) \end{bmatrix}. \tag{19}$$

Since each x corresponds to some y and vice versa, it also holds for the z-transforms of arbitrary sequences $(x^0, x^1, \dots)$ and corresponding $(y^0, y^1, \dots)$ using the same 2×2 matrix.

Proposition 4. *(Special cases of linear relations)*

1. Identity: *If $\phi_1 = \phi_2$, then $\phi_1 \overset{\hat{M}}{\sim} \phi_2$ with $\hat{M} = \left[\begin{smallmatrix} I & 0 \\ 0 & I \end{smallmatrix}\right]$.*
2. Commutation: *If $\phi(\hat{C}\hat{y}) = \hat{C}\phi(\hat{y})$ for all $\hat{y}$, then $\phi \overset{\hat{M}}{\sim} \phi$ with $\hat{M} = \begin{bmatrix} \hat{C} & 0 \\ 0 & \hat{C} \end{bmatrix}$.*
3. Equivariance: *If $\phi_1(\hat{A}\hat{y}) = \hat{B}\phi_2(\hat{y})$ for all $\hat{y}$, then $\phi_1 \overset{\hat{M}}{\sim} \phi_2$ with $\hat{M} = \begin{bmatrix} \hat{A} & 0 \\ 0 & \hat{B} \end{bmatrix}$.*
4. Concatenation: *If $\psi_i \overset{\hat{M}_i}{\sim} \phi_i$ with $\hat{M}_i = \begin{bmatrix} \hat{P}_i & \hat{Q}_i \\ \hat{R}_i & \hat{S}_i \end{bmatrix}$ for $i = 1, \dots, p$, then $(\psi_1, \dots, \psi_p) \overset{\hat{M}'}{\sim} (\phi_1, \dots, \phi_p)$ with $\hat{M}' = \begin{bmatrix} \mathbf{diag}(\hat{P}_i) & \mathbf{diag}(\hat{Q}_i) \\ \mathbf{diag}(\hat{R}_i) & \mathbf{diag}(\hat{S}_i) \end{bmatrix}$.*

When $\Phi_1 \overset{\hat{M}}{\sim} \Phi_2$ and these oracles are used with algorithms $\hat{H}_1$ and $\hat{H}_2$, respectively, we must have $\hat{y}_1 = \hat{H}_1 \hat{u}_1$ and $\hat{y}_2 = \hat{H}_2 \hat{u}_2$. Incorporating this with Definition 6, we can define a natural generalization of equivalence that holds in this setting.

Definition 7 (LFT equivalence)**.** Consider $\hat{H}_1 \circ \Phi_1$ and $\hat{H}_2 \circ \Phi_2$, where $\Phi_1 \overset{\hat{M}}{\sim} \Phi_2$. We say the algorithms are *LFT-equivalent* and write $\hat{H}_1 \circ \Phi_1 \overset{\hat{M}}{\sim} \hat{H}_2 \circ \Phi_2$, if

$$\mathcal{R} \begin{bmatrix} \hat{H}_1 \\ I \end{bmatrix} = \hat{M}\, \mathcal{R} \begin{bmatrix} \hat{H}_2 \\ I \end{bmatrix}. \tag{20}$$

We justify the name "LFT" in Remark 4 and "equivalence" in Remark 3.

We omit $\hat{M}$ and simply write $\hat{H}_1 \circ \Phi_1 \sim \hat{H}_2 \circ \Phi_2$ when $\hat{M}$ is the same as that for which $\Phi_1 \overset{\hat{M}}{\sim} \Phi_2$, and therefore clear from context.

Remark 3. Linear equivalence $\Phi_1 \sim \Phi_2$ and LFT equivalence $\hat{H}_1 \circ \Phi_1 \sim \hat{H}_2 \circ \Phi_2$ satisfy *reflexivity* and *symmetry*, and *transitivity*, so they are *equivalence relations*.

Our main result of this section is an algebraic characterization of LFT equivalence between algorithms defined in Definition 7.

Theorem 5 (algebraic characterization of LFT equivalence)**.** *Suppose* $\Phi_1 \overset{\hat{M}}{\sim} \Phi_2$*. Then* $\hat{H}_1 \circ \Phi_1 \overset{\hat{M}}{\sim} \hat{H}_2 \circ \Phi_2$ *if and only if*

$$\begin{bmatrix} I & -\hat{H}_1 \end{bmatrix} \hat{M} \begin{bmatrix} \hat{H}_2 \\ I \end{bmatrix} = 0, \quad \textit{or equivalently,} \quad \begin{bmatrix} I & -\hat{H}_2 \end{bmatrix} \hat{M}^{-1} \begin{bmatrix} \hat{H}_1 \\ I \end{bmatrix} = 0.$$

Proof. See Appendix B.2. □

We can apply Theorem 5 to solve for $\hat{H}_1$ in terms of $\hat{H}_2$ or vice versa.

Corollary 6. *Consider the setting of Theorem 5 with* $\hat{M} = \left[\begin{smallmatrix} \hat{P} & \hat{Q} \\ \hat{R} & \hat{S} \end{smallmatrix}\right]$*. Then we have* $\hat{H}_1(\hat{R}\hat{H}_2 + \hat{S}) = (\hat{P}\hat{H}_2 + \hat{Q})$*. In particular,*

$$\hat{H}_1 = (\hat{P}\hat{H}_2 + \hat{Q})(\hat{R}\hat{H}_2 + \hat{S})^{-1} \quad \textit{and} \quad \hat{H}_2 = (-\hat{H}_1\hat{R} + \hat{P})^{-1}(\hat{H}_1 S - \hat{Q}). \tag{21}$$

Remark 4. The relationships between $\hat{H}_1$ and $\hat{H}_2$ in Eq. (21) are commonly called *linear fractional transformations* (LFTs), which is why we chose the name *LFT equivalence*.

The results above can also be derived by direct manipulation of the block diagram as we demonstrated with shift equivalence in Fig. 5. In this case, the manipulation is a bit more involved; see Fig. 6.

The dashed box in Fig. 6 represents the equivalent $\hat{H}_2$. Based on the block diagram, we obtain the following algebraic relationships:

$$\hat{y}_2 = \begin{bmatrix} I & 0 \end{bmatrix} \hat{M}^{-1} \begin{bmatrix} \hat{H}_1 \\ I \end{bmatrix} \begin{bmatrix} 0 & I \end{bmatrix} \hat{M} \begin{bmatrix} \hat{y}_2 \\ \hat{u}_2 \end{bmatrix} \qquad \text{and} \qquad \hat{y}_2 = \hat{H}_2 \hat{u}_2.$$

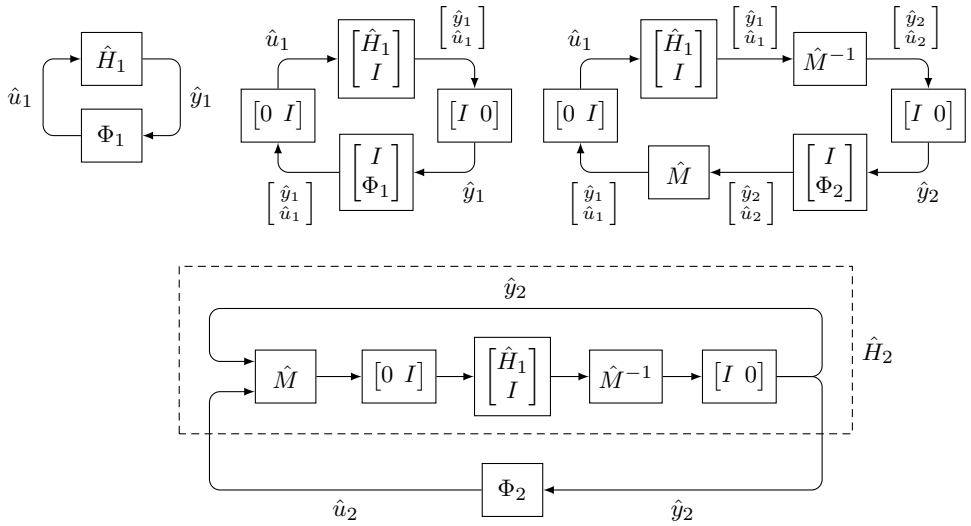

Fig. 6: Equivalent block diagrams representing LFT equivalence. Starting from the top left, we augment the algorithm and oracle, we transform the oracle using the linear equivalence $\Phi_1 \overset{\hat{M}}{\sim} \Phi_2$, and finally we isolate the equivalent $\hat{H}_2$ in feedback with Φ_2.

These equations can be resolved in various ways. Most relevant for our purpose, we eliminate $\hat{y}_2$ and seek an identity that holds for all $\hat{u}_2$, which leads to:

$$\hat{H}_2 = \begin{bmatrix} I & 0 \end{bmatrix} \hat{M}^{-1} \begin{bmatrix} \hat{H}_1 \\ I \end{bmatrix} \begin{bmatrix} 0 & I \end{bmatrix} \hat{M} \begin{bmatrix} \hat{H}_2 \\ I \end{bmatrix}.$$

This expression can be further simplified to obtain the relationships in Theorem 5 and Corollary 6.

The special cases of Proposition 4 lead to simple expressions for LFT equivalence between algorithms.

Corollary 7 (commutation)**.** *Suppose $\phi(\hat{C}\hat{y}) = \hat{C}\phi(\hat{y})$ for all $\hat{y}$. Then $\hat{H}_1 \circ \phi \overset{\hat{M}}{\sim} \hat{H}_2 \circ \phi$ with $\hat{M} = \begin{bmatrix} \hat{C} & 0 \\ 0 & \hat{C} \end{bmatrix}$. Consequently, $\hat{H}_1 = \hat{C}\hat{H}_2\hat{C}^{-1}$. If we let $\hat{C} = \hat{\Delta}_m$ (multi-shift), then we see that shift equivalence is a special case of LFT equivalence.*

Corollary 8 (equivariance)**.** *Suppose $\phi_1(\hat{A}\hat{y}) = \hat{B}\phi_2(\hat{y})$ for all $\hat{y}$. Then $\hat{H}_1 \circ \phi_1 \overset{\hat{M}}{\sim} \hat{H}_2 \circ \phi_2$ with $\hat{M} = \begin{bmatrix} \hat{A} & 0 \\ 0 & \hat{B} \end{bmatrix}$. Consequently, $\hat{H}_1 = \hat{A}\hat{H}_2\hat{B}^{-1}$.*

We are also interested in the special case of algorithms $\hat{H}_1 \circ (\Psi, \phi_1)$ and $\hat{H}_2 \circ (\Psi, \phi_2)$, where Ψ is some set of oracles common to both algorithms, and $\phi_1 \overset{\hat{M}}{\sim} \phi_2$ with $\hat{M} =$

$\begin{bmatrix} \hat{P} & \hat{Q} \\ \hat{R} & \hat{S} \end{bmatrix}$ In this case, by concatenation (Proposition 4), we have $(\Psi, \phi_1) \overset{\hat{M}'}{\sim} (\Psi, \phi_2)$ with $\hat{M}' = \begin{bmatrix} I & 0 & 0 & 0 \\ 0 & \hat{P} & 0 & \hat{Q} \\ 0 & 0 & I & 0 \\ 0 & \hat{R} & 0 & \hat{S} \end{bmatrix}$. Therefore, we immetiately obtain the following corollary.

Corollary 9 (LFT equivalence with common oracles)**.** *Suppose* $\phi_1 \overset{\hat{M}}{\sim} \phi_2$ *with* $\hat{M} = \begin{bmatrix} \hat{P} & \hat{Q} \\ \hat{R} & \hat{S} \end{bmatrix}$*. Let* Ψ *be another oracle. Then* $\hat{H}_1 \circ (\Psi, \phi_1) \overset{\hat{M}'}{\sim} \hat{H}_2 \circ (\Psi, \phi_2)$ *with* $\hat{M}'$ *defined above, if (see Corollary 6):* $\hat{H}_1 \left(\begin{bmatrix} 0 & 0 \\ 0 & \hat{R} \end{bmatrix} \hat{H}_2 + \begin{bmatrix} I & 0 \\ 0 & \hat{S} \end{bmatrix} \right) = \begin{bmatrix} I & 0 \\ 0 & \hat{P} \end{bmatrix} \hat{H}_2 + \begin{bmatrix} 0 & 0 \\ 0 & \hat{Q} \end{bmatrix}$.

Efficient determination of LFT equivalence

To determine whether $\hat{H}_1 \circ \Phi_1 \sim \hat{H}_2 \circ \Phi_2$, we must first establish how Φ_1 and Φ_2 are related. If there exists some $\hat{M}$ such that $\Phi_1 \overset{\hat{M}}{\sim} \Phi_2$, then we can apply Theorem 5, and we have $\hat{H}_1 \circ \Phi_1 \sim \hat{H}_2 \circ \Phi_2$ if $\begin{bmatrix} I & -\hat{H}_1 \end{bmatrix} \hat{M} \begin{bmatrix} \hat{H}_2 \\ I \end{bmatrix} = 0$. For an example of how this result can be used in practice, see the end of Section 8.2.

If there are many different $\hat{M}$ matrices that work, say $\Phi_1 \overset{\hat{M}}{\sim} \Phi_2$ for all $\hat{M} \in \mathcal{M}$, then it follows from Definition 6 that $\mathcal{M}$ is a multiplicative group. Determining equivalence amounts to checking feasiblity of the problem $\begin{bmatrix} I & -\hat{H}_1 \end{bmatrix} \hat{M} \begin{bmatrix} \hat{H}_2 \\ I \end{bmatrix} = 0$ with $\hat{M} \in \mathcal{M}$. We saw at the end of Section 7 how to solve this problem for the special case of shift-equivalence ($\mathcal{M}$ is the set of multi-shifts). However, we suspect that solving this problem for different $\mathcal{M}$ would require a case-by-case analysis.

8.1 Proxes, subdifferentials, and their conjugates

We are now ready to return to the motivating examples of Algorithms 8.1–8.4. The oracles $\{\partial f, \partial f^*, \mathrm{prox}_{tf}, \mathrm{prox}_{\frac{1}{t} f^*}\}$ are linearly equivalent to one another. Using the identities at the beginning of Section 8, these relationships can be derived as in Eq. (19). The associated matrices $\hat{M}$ corresponding to Definition 6 are given in Fig. 7.

Note that the diagonal entries of Fig. 7 are *identity LFT matrices* (see Proposition 4). Applying Corollary 9 to the matrices in Fig. 7, we can obtain a set of algorithms equivalent when we swap one oracle for another.

Corollary 10 (LFT equivalence for prox)**.** *Suppose* $\hat{H}$ *is an algorithm that uses oracles partitioned as* $(\Psi, \mathrm{prox}_{tg})$*. Then, the following transfer functions correspond to LFT-equivalent algorithms.*

$$\begin{bmatrix} \hat{H}_{11} & \hat{H}_{12} \\ \hat{H}_{21} & \hat{H}_{22} \end{bmatrix} \circ (\Psi, \mathrm{prox}_{tg}),$$

$$\begin{bmatrix} \hat{H}_{11} + \hat{H}_{12}(I - \hat{H}_{22})^{-1}\hat{H}_{21} & -t\hat{H}_{12}(I - \hat{H}_{22})^{-1} \\ \frac{1}{t}(I - \hat{H}_{22})^{-1}\hat{H}_{21} & -\hat{H}_{22}(I - \hat{H}_{22})^{-1} \end{bmatrix} \circ (\Psi, \mathrm{prox}_{\frac{1}{t} g^*}),$$

$$\begin{bmatrix} \hat{H}_{11} + \hat{H}_{12}(I - \hat{H}_{22})^{-1}\hat{H}_{21} & -t\hat{H}_{12}(I - \hat{H}_{22})^{-1} \\ (I - \hat{H}_{22})^{-1}\hat{H}_{21} & -t(I - \hat{H}_{22})^{-1} \end{bmatrix} \circ (\Psi, \partial g),$$

$\phi_1 \backslash \phi_2$	∂f	∂f^*	prox_{tf}	$\mathrm{prox}_{\frac{1}{t}f^*}$
∂f	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ \frac{1}{t} & -\frac{1}{t} \end{bmatrix}$	$\begin{bmatrix} t & -t \\ 0 & 1 \end{bmatrix}$
∂f^*	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \frac{1}{t} & -\frac{1}{t} \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ t & -t \end{bmatrix}$
prox_{tf}	$\begin{bmatrix} 1 & t \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} t & 1 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} t & 0 \\ t & -t \end{bmatrix}$
$\mathrm{prox}_{\frac{1}{t}f^*}$	$\begin{bmatrix} \frac{1}{t} & 1 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & \frac{1}{t} \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} \frac{1}{t} & 0 \\ \frac{1}{t} & -\frac{1}{t} \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

Fig. 7: Matrices $\hat{M}$ conforming to Definition 6 for all possible linear equivalences between the oracles $\{\partial f, \partial f^*, \mathrm{prox}_{tf}, \mathrm{prox}_{\frac{1}{t}f^*}\}$.

$$\begin{bmatrix} \hat{H}_{11} & \hat{H}_{12} \\ \frac{1}{t}\hat{H}_{21} & -\frac{1}{t}(I - \hat{H}_{22}) \end{bmatrix} \circ (\Psi, \partial g^*).$$

Corollary 11 (LFT equivalence for subdifferentials)**.** *Suppose $\hat{H}$ is an algorithm that uses oracles partitioned as $(\Psi, \partial g)$. Then, the following transfer functions correspond to LFT-equivalent algorithms.*

$$\begin{bmatrix} \hat{H}_{11} & \hat{H}_{12} \\ \hat{H}_{21} & \hat{H}_{22} \end{bmatrix} \circ (\Psi, \partial g)$$
$$\begin{bmatrix} \hat{H}_{11} - \hat{H}_{12}\hat{H}_{22}^{-1}\hat{H}_{21} & \hat{H}_{12}\hat{H}_{22}^{-1} \\ -\hat{H}_{22}^{-1}\hat{H}_{21} & \hat{H}_{22}^{-1} \end{bmatrix} \circ (\Psi, \partial g^*)$$
$$\begin{bmatrix} \hat{H}_{11} - \hat{H}_{12}\hat{H}_{22}^{-1}\hat{H}_{21} & \hat{H}_{12}\hat{H}_{22}^{-1} \\ -t\hat{H}_{22}^{-1}\hat{H}_{21} & I + t\hat{H}_{22}^{-1} \end{bmatrix} \circ (\Psi, \mathrm{prox}_{tg})$$
$$\begin{bmatrix} \hat{H}_{11} & \hat{H}_{12} \\ \frac{1}{t}\hat{H}_{21} & \frac{1}{t}\hat{H}_{22} + I \end{bmatrix} \circ (\Psi, \mathrm{prox}_{\frac{1}{t}g^*})$$

Remark 5. If there is no Ψ in Corollaries 10 and 11 (the prox or subdifferential is the only oracle), then we can extract the $(2,2)$ blocks of all submatrices and we obtain LFT-equivalence among:

$$\hat{H} \circ \mathrm{prox}_{tg}, \quad -\hat{H}(I - \hat{H})^{-1} \circ \mathrm{prox}_{\frac{1}{t}g^*}, \quad -t(I - \hat{H})^{-1} \circ \partial g, \quad -\tfrac{1}{t}(I - \hat{H}) \circ \partial g^*$$

and among: $\hat{H} \circ \partial g$, $\hat{H}^{-1} \circ \partial g^*$, $(I + t\hat{H}^{-1}) \circ \mathrm{prox}_{tg}$, $(\frac{1}{t}\hat{H} + I) \circ \mathrm{prox}_{\frac{1}{t}g^*}$.

Remark 6. In Corollary 11, $\hat{H}_{22}$ must be invertible. We may want to also ensure that $\hat{H}_{22}^{-1}$ is proper, as this is necessary if we want an implementable algorithm. The condition that a transfer function $\hat{H}$ be invertible and proper can be characterized precisely [49, Lem. 3.15]; it is equivalent to requiring that $D = \lim_{z\to\infty} \hat{H}(z)$ is invertible. One possible state-space realization of the inverse transfer function $\hat{H}^{-1}$ is

$$\hat{H}^{-1} = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]^{-1} = \left[\begin{array}{c|c} A - BD^{-1}C & BD^{-1} \\ \hline -D^{-1}C & D^{-1} \end{array}\right].$$

8.2 Examples of LFT equivalence

Algorithms 8.1–8.4

We can verify equivalence of Algorithms 8.1–8.4 by directly applying Corollary 10. Specifically, substituting $\hat{H} = \begin{bmatrix} 0 & \frac{1}{z} \\ -t & \frac{1}{z} \end{bmatrix}$ in Corollary 10, we immediately obtain the transfer functions in Eq. (18).

Oracle swapping and deletion

Although we did not cover oracle swapping or deletion, both of these notions are trivial to check in our framework. In fact, they are special cases of LFT equivalence.

Most splitting methods are not symmetric with respect to oracle swapping since the different oracles usually have different properties that the algorithm is trying to exploit. Nevertheless, one might be interested in, e.g., Davis–Yin splitting where f and g are swapped (see Algorithm 8.6). This is called *dual Davis–Yin* in [45]. Given $\hat{H} \circ \Psi$, if Ψ is a permutation of the oracles Φ, then we can write $\Psi(Px) = P\Phi(x)$, where P is a permutation matrix. By Corollary 8, we have $\hat{H} \circ \Psi = P^\top \hat{H} P \circ \Phi$, i.e., permute the corresponding rows and columns of the transfer matrix. We exploit this fact later in this section when we show that PD3O is LFT-equivalent to Davis–Yin splitting.

DR and Chambolle–Pock

We can use LFT equivalence to show the relation between Douglas–Rachford (DR), Algorithm 3.5, and the primal-dual optimization method proposed by Chambolle and Pock (Algorithm 8.5 [18, 50]).

Algo. 8.5 Chambolle–Pock

for $k = 0, 1, 2, \ldots$ **do**
  $x_1^{k+1} = \operatorname{prox}_{\tau f}(x_1^k - \tau M^\top x_2^k)$
  $x_2^{k+1} = \operatorname{prox}_{\sigma g^*}(x_2^k + \sigma M(2x_1^{k+1} - x_1^k))$
end for

Comparing Algorithm 8.5 and Algorithm 3.5, we should first set $\tau = \sigma = 1$ so that the oracles correspond properly. Now, computing transfer functions, we have:

$$\text{Algorithm 3.5: } \begin{bmatrix} \frac{-1}{z-1} & \frac{1}{z-1} \\ \frac{2z-1}{z-1} & \frac{-1}{z-1} \end{bmatrix} \circ (\operatorname{prox}_f, \operatorname{prox}_g) \quad \text{and}$$

$$\text{Algorithm 8.5: } \begin{bmatrix} \frac{1}{z} & -\frac{1}{z}M^\top \\ \frac{2z-1}{z}M & \frac{1}{z} \end{bmatrix} \circ (\text{prox}_f, \text{prox}_{g^*}).$$

Applying Corollary 10, we will have LFT-equivalence between these algorithms if

$$\begin{bmatrix} \frac{1}{z} & -\frac{1}{z}M^\top \\ \frac{2z-1}{z}M & \frac{1}{z} \end{bmatrix} = \begin{bmatrix} \frac{1}{z} & -\frac{1}{z} \\ \frac{2z-1}{z} & \frac{1}{z} \end{bmatrix}$$

Therefore, Algorithms 3.5 and 8.5 are LFT-equivalent if $M = I$.

More three-operator splitting

An algorithm that has recently attracted considerable attention is the three-operator splitting algorithm of Davis and Yin [51]. This algorithm solves the problem

$$\text{minimize } f(x) + g(x) + h(x)$$

using the oracles prox_f, prox_g, and ∇h. The algorithm and its transfer function are given as follows.

Algo. 8.6 Davis–Yin three-operator splitting

for $k = 0, 1, 2, \ldots$ **do**
  $z^k = \text{prox}_{tf}(y^k)$
  $x^k = \text{prox}_{tg}(2z^k - y^k - t\nabla h(z^k))$
  $y^{k+1} = y^k - z^k + x^k$
end for

$$\hat{H}_{8.6}(z) = \begin{bmatrix} \frac{-1}{z-1} & \frac{1}{z-1} & 0 \\ \frac{2z-1}{z-1} & \frac{-1}{z-1} & -t \\ 1 & 0 & 0 \end{bmatrix}$$

Suppose we wanted to design an equivalent algorithm that used the oracles $(\text{prox}_{tf}, \text{prox}_{g^*}, \nabla h)$ instead. We proceed in steps:

$$\begin{bmatrix} \frac{-1}{z-1} & \frac{1}{z-1} & 0 \\ \frac{2z-1}{z-1} & \frac{-1}{z-1} & -t \\ 1 & 0 & 0 \end{bmatrix} \circ (\text{prox}_{tf}, \text{prox}_{tg}, \nabla h)$$

$$\sim \begin{bmatrix} \frac{-1}{z-1} & 0 & \frac{1}{z-1} \\ 1 & 0 & 0 \\ \frac{2z-1}{z-1} & -t & \frac{-1}{z-1} \end{bmatrix} \circ (\text{prox}_{tf}, \nabla h, \text{prox}_{tg}) \qquad \begin{pmatrix} \text{swap last two rows} \\ \text{and columns} \end{pmatrix}$$

$$\sim \begin{bmatrix} \frac{1}{z} & -\frac{t}{z} & -\frac{t}{z} \\ 1 & 0 & 0 \\ \frac{2z-1}{tz} & -\frac{(z-1)}{z} & \frac{1}{z} \end{bmatrix} \circ (\text{prox}_{tf}, \nabla h, \text{prox}_{\frac{1}{t}g^*}) \qquad \begin{pmatrix} \text{apply Corollary 10} \\ \text{with } \Phi = (\text{prox}_{tf}, \nabla h) \end{pmatrix}$$

$$\sim \begin{bmatrix} \frac{1}{z} & -\frac{t}{z} & -\frac{t}{z} \\ \frac{2z-1}{tz} & \frac{1}{z} & -\frac{(z-1)}{z} \\ 1 & 0 & 0 \end{bmatrix} \circ (\text{prox}_{tf}, \text{prox}_{\frac{1}{t}g^*}, \nabla h) \qquad \begin{pmatrix} \text{swap last two rows} \\ \text{and columns} \end{pmatrix}$$

Now, compare this algorithm to PD3O (Algorithm 7.1), which is

$$\begin{bmatrix} \frac{1}{z} & \frac{-\tau A^\top}{z} & \frac{-\tau}{z} \\ \frac{\sigma(2z-1)A}{z} & \frac{1}{z} & \frac{-\sigma\tau(z-1)A}{z} \\ 1 & 0 & 0 \end{bmatrix} \circ (\mathrm{prox}_{tf}, \mathrm{prox}_{\sigma g^*}, \nabla h)$$

We can see that the algorithms are LFT-equivalent upon setting $A = I$, $\tau = t$, $\sigma = \frac{1}{t}$. Although this result is known [43, 45], the benefit of systematizing algorithm equivalence is that these sorts of equivalences can be determined straightforwardly. We can directly verify the equivalence above by applying Theorem 5 with $\hat{M} = \left[\begin{smallmatrix} t & 0 \\ t & -t \end{smallmatrix}\right]$ taken from Fig. 7 and applied to the middle oracle to transform prox_{tg} to $\mathrm{prox}_{\frac{1}{t}g^*}$:

$$\begin{bmatrix} I & -\hat{H}_{8.6} \end{bmatrix} \hat{M} \begin{bmatrix} \hat{H}_{7.1} \\ I \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \frac{1}{z-1} & \frac{-1}{z-1} & 0 \\ 0 & 1 & 0 & \frac{1-2z}{z-1} & \frac{1}{z-1} & t \\ 0 & 0 & 1 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & t & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & t & 0 & 0 & -t & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{z} & \frac{-t}{z} & \frac{-t}{z} \\ \frac{2z-1}{tz} & \frac{1}{z} & \frac{1-z}{z} \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = 0.$$

9 Discussion

One algorithm, many interpretations and implementations

Is it useful to have many different forms of an algorithm, if all the forms are LFT-equivalent? Yes: different rewritings of one algorithm often yield different ("physical") intuition. For example, Algorithm 1.1 uses the current loss function for extrapolation [52]; while Algorithm 1.2 seems to extrapolate from the previous loss function [53]. The distributed algorithms Algorithms 6.7 and 6.8, although equivalent, were developed using very different intuition. The former used a *gradient differencing scheme* [38] whereas the latter used an *adapt-correct-combine* approach [39].

Equivalent algorithms can differ in memory usage, computational efficiency, or numerical stability. For example, implementations of Algorithms 1.3 and 1.4 lead to different memory usage [5, 6]. At each time step k, Algorithm 1.3 needs to store x_2^{k+1} and $F(x_2^k)$, but Algorithm 1.4 only needs to store x_1^k in memory. These different rewritings also naturally yield different generalizations, for example, by projecting different state variables. Likewise, Douglas–Rachford (Algorithm 3.5) only requires storing x_3^k at each time step k, whereas simplified ADMM (Algorithm 3.6) requires storing ξ_2^k and ξ_3^k. This is evident from Fig. 3; the dotted lines cross one arrow for Douglas–Rachford and two arrows for ADMM.

Stochastic and randomized algorithms

Our framework applies to stochastic or randomized algorithms with almost no modifications, simply by allowing random oracles. For example, we can accept oracles like random search $\mathrm{argmin}\{f(x+\omega_i) : i = 1, \ldots, k\}$, stochastic gradient $\nabla f(x)+\omega$, or noisy gradient $\nabla f(x+\omega)$. The definition of oracle equivalence requires a slight modification in this setting: for algorithms that use (pseudo-)randomized oracles, two algorithms

are oracle-equivalent if they generate identical sequences of oracle calls given the same random seed.

Time-varying algorithms

The linear time-invariant (LTI) algorithm assumption is critical, as the ability to relate the z-transforms of the input and output via multiplication with a transfer function ($\hat{y} = \hat{H}\hat{u}$) critically relies on the map $(u^0, u^1, \dots) \mapsto (y^0, y^1, \dots)$ being LTI.

Nevertheless, many of the other concepts from Section 5 do extend to systems that are time varying. For example, an algorithm with parameters that change on a fixed schedule but is otherwise linear, such as gradient descent with a diminishing stepsize, can be regarded as a linear time-varying (LTV) system [29], and the notion of a transfer function has been generalized to LTV systems [54]. If, instead, the parameters change adaptively based on the other state variables, the system can be regarded as a linear parameter varying (LPV) system [55] or a switched system [56]. Examples of such algorithms include nonlinear conjugate gradient methods and quasi-Newton methods.

Oracle structure

We assumed throughout this paper that all oracles were *nonlinear and time-invariant*. If we weaken this assumption, and let the oracles be *nonlinear and time-varying*, the notion of oracle equivalence is still meaningful: it holds if the two algorithms invoke the same sequence of oracle calls. However, shift equivalence no longer works because time-varying operators do not commute with time shifts.

If we strengthen the assumption instead, and assume the oracles are endowed with additional structure, then further equivalences are possible. Indeed, every commutation relation satisfied by the oracle leads to a new notion of equivalence!

For example, an oracle that is *linear and time-invariant* would commute with any other LTI system (not just multi-shifts). As an example, consider DR (Algorithm 3.5) where f is known to be a quadratic function. In this case, the oracle $L = \mathrm{prox}_f$ is *linear* and therefore commutes with *any LTI system*. For example, it commutes with the dynamical system:

$$\left\{\begin{array}{l} x^{k+1} = x^k - \alpha u^k \\ y^k = x^k \end{array}\right\} \quad = \quad \left[\begin{array}{c|c} 1 & -\alpha \\ \hline 1 & 0 \end{array}\right] \quad = \quad \frac{1}{z-\alpha}$$

Assuming $x^0 = 0$, this dynamical system maps $(u^0, u^1, \dots) \mapsto (y^0, y^1, \dots)$, with

$$y^k = u^k + \alpha u^{k-1} + \alpha^2 u^{k-2} + \cdots + \alpha^k u^0 \qquad \text{for } k = 0, 1, \dots \tag{22}$$

This transformation clearly commutes with a linear oracle L, because left-multiplying each u^k by L and then applying the transformation (22) is the same as applying (22) first and then left-multiplying by L. In other words,

$$L \cdot \tfrac{1}{z-\alpha} \cdot \hat{y} = \tfrac{1}{z-\alpha} \cdot L \cdot \hat{y}.$$

Since DR uses oracles $(\mathrm{prox}_f, \mathrm{prox}_g)$ and only prox_f is assumed to be linear, the special commutation relation only holds for prox_f, and we may write

$$\begin{bmatrix} \mathrm{prox}_f & 0 \\ 0 & \mathrm{prox}_g \end{bmatrix} \begin{bmatrix} \frac{1}{z-\alpha} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{z-\alpha} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathrm{prox}_f & 0 \\ 0 & \mathrm{prox}_g \end{bmatrix} \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix}.$$

So when f is a quadratic function, we have via Corollary 7 that $\hat{H}_1 \circ (\mathrm{prox}_f, \mathrm{prox}_g) \sim \hat{H}_2 \circ (\mathrm{prox}_f, \mathrm{prox}_g)$ if $\hat{H}_2 = \left[\begin{smallmatrix} \frac{1}{z-\alpha} & 0 \\ 0 & 1 \end{smallmatrix}\right]^{-1} \hat{H}_1 \left[\begin{smallmatrix} \frac{1}{z-\alpha} & 0 \\ 0 & 1 \end{smallmatrix}\right]$. Letting $\hat{H}_1 = \hat{H}_{3.5}$, we obtain the equivalent algorithm:

$$\begin{bmatrix} \frac{-1}{z-1} & \frac{1}{z-1} \\ \frac{2z-1}{z-1} & \frac{-1}{z-1} \end{bmatrix} \circ (\mathrm{prox}_f, \mathrm{prox}_g) \sim \begin{bmatrix} \frac{-1}{z-1} & \frac{z-\alpha}{z-1} \\ \frac{2z-1}{(z-1)(z-\alpha)} & \frac{-1}{z-1} \end{bmatrix} \circ (\mathrm{prox}_f, \mathrm{prox}_g)$$

One possible realization of the new algorithm is given below.

Algo. 9.1 Quadratic-f variant of DR

for $k = 0, 1, \dots$ **do**
  $y^k = \mathrm{prox}_g(x_2^k - 2x_1^k)$
  $x_1^{k+1} = \alpha x_1^k - \mathrm{prox}_f\big(\alpha x_2^k - x_2^{k+1}\big)$
  $x_2^{k+1} = x_2^k - x_1^k - y^k$
end for

Therefore, Algorithms 3.5 and 9.1 are equivalent for all α when f is a quadratic function, but they cease to be equivalent when we remove this constraint on f. Specific equivalence results that require one of the oracles to be linear can be found, for example, in [57, Theorem 4]. However, the approach presented above is far more general, as it allows one to systematically derive entire families of equivalent algorithms.

Moving beyond linearity, different notions of equivalence could conceivably be developed for other classes of oracles, such as dynamic oracles (oracles with memory), or multi-dimensional oracles that have structure, such as sparsity.

Nonlinear state updates

Our main exposition only considers algorithms defined by state-space equations: a linear map relates (x^k, u^k) to (x^{k+1}, y^k). However, this assumption can be relaxed: linear state updates is a sufficient condition, but it is not necessary. We only require that the map $(u^0, u^1, \dots) \mapsto (y^0, y^1, \dots)$ be LTI. For example, consider Algorithm 9.2, which is related to ordinary gradient descent (Algorithm 3.4) via a nonlinear state transformation.

Algo. 9.2

for $k = 0, 1, 2, \dots$ **do**
  $x^{k+1} = x^k \exp(-\frac{1}{5}\nabla f(\log x^k))$
end for

Although the state update equations for Algorithm 9.2 are nonlinear, if we identify the oracle input $y^k = \log x^k$ and the oracle output $u^k = \nabla f(y^k)$, we can eliminate x^k

and write the algorithm as $y^{k+1} = y^k - \frac{1}{5}u^k$, which is a linear system with transfer function $-\frac{1}{5}\frac{1}{z-1}$.

Checking for equivalence

Checking for equivalence in our framework is straightforward using a computer algebra system to verify relations between transfer functions, as demonstrated in Section 5. This process can be automated using symbolic computation libraries. A previous version of our framework has been implemented in the software package LINNAEUS, which allows researchers to search for algorithms that are related by oracle and shift equivalence, or a weaker form of LFT-equivalence [58].

10 Conclusion

In this paper, we have presented a framework for reasoning about equivalence between a broad class of iterative algorithms by using ideas from control theory to represent optimization algorithms. The main insight is that by representing an algorithm as a linear dynamical system in feedback with a static nonlinearity, we can recognize equivalent algorithms by detecting algebraic relations between the transfer functions of the associated linear systems. This framework can identify algorithms that result in the same sequence of oracle calls (up to a possible shift), even when algorithms use different but related oracles.

Our main framework requires that the algorithm is linear in the state and oracle outputs, but not necessarily in the parameters. Our work focused on the case where the oracles are nonlinear and time-invariant, but these assumptions can be strengthened or weakened, which would modify the corresponding notions of algorithm equivalence in a predictable and principled manner, following directly from the framework we have developed. We have discussed how to use our framework to understand distributed algorithms and randomized algorithms, and how to extend our framework to encompass nonlinear state transformations or time-varying oracles.

Our work presents first steps towards systematizing the study of optimization algorithms. When viewed as dynamical systems and characterized in terms of their input-output maps, algorithms are distilled to their essential function: a causal map that produces the next oracle input based on past oracle outputs.

Looking forward, *control theory* is well-positioned to advance the fields of algorithm discovery, analysis, and design. Control theory is concerned with the analysis and synthesis of dynamical systems with the goal of obtaining desirable overall behavior, such as stability or robustness to noise. In particular, tools from *robust control* have been used to analyze and design optimization algorithms with optimized convergence rates or noise-robustness properties, for example [12, 59].

Appendix A Control theory results

A.1 Minimal realizations

We start with the important definitions of *controllability*, *observability*, and *minimality*.

Definition 8. Consider a realization (A, B, C, D) with $A \in \mathbb{R}^{n\times n}$. The realization, or simply the pair (A, B), is *controllable* if the controllability matrix $\mathcal{C}$ has full row rank. The realization, or simply the pair (C, A), is *observable* if the observability matrix $\mathcal{O}$ has full column rank. The controllability and observability matrices are defined as:

$$\mathcal{C} := \begin{bmatrix} B & AB & \cdots & A^{n-1}B \end{bmatrix} \quad \text{and} \quad \mathcal{O} := \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{n-1} \end{bmatrix}.$$

Definition 9. A realization (A, B, C, D) of $\hat{H}(z)$ is said to be *minimal* if A has the smallest possible dimension.

The notions of controllability and observability are intimately connected to the notion of a minimal realization [49, §3.7].

Proposition 12. *A realization is minimal if and only if it is controllable and observable. Furthermore, all minimal realizations of $\hat{H}(z)$ are related to one another via a suitably chosen invertible matrix T and the transformation (14).*

The transfer function corresponding to a realization (A, B, C, D) can be expanded into an infinite series. Its (matrix) coefficients M_k are called the *Markov parameters* and are defined as:

$$\begin{aligned} \hat{H}(z) &= D + C(zI - A)^{-1}B \\ &= D + CBz^{-1} + CABz^{-2} + CA^2Bz^{-3} + \cdots + CA^{k-1}Bz^{-k} + \cdots \\ &=: M_0 + M_1z^{-1} + M_2z^{-2} + M_3z^{-3} + \cdots + M_kz^{-k} + \cdots \end{aligned}$$

The Markov parameters only depend on the transfer function, so all realizations of a given transfer function have the same Markov parameters. We can arrange the Markov parameters into the semi-infinite *Hankel matrix*, which factors as:

$$\mathbf{H} := \begin{bmatrix} M_1 & M_2 & M_3 & \cdots \\ M_2 & M_3 & M_4 & \cdots \\ M_3 & M_4 & M_5 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} = \begin{bmatrix} CB & CAB & CA^2B & \cdots \\ CAB & CA^2B & CA^3B & \cdots \\ CA^2B & CA^3B & CA^4B & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} = \underbrace{\begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \end{bmatrix}}_{\mathbf{O}} \underbrace{\begin{bmatrix} B & AB & A^2B & \cdots \end{bmatrix}}_{\mathbf{C}}.$$

In particular, this identity must hold for a minimal realization, which is controllable and observable by Proposition 12. We immediately obtain the following result.

Proposition 13. *The minimal realization of a system has a state dimension equal to* $n = \mathbf{rank}(\mathbf{H})$*, where* $\mathbf{H}$ *is the Hankel matrix of the system.*

Remark 7. When applying Proposition 13 in practice, it suffices to compute a block-$N \times N$ truncation of $\mathbf{H}$, where N is any upper bound on the minimal state dimension. For example, we can let N be the dimension of any given realization. This works because due to the Cayley–Hamilton theorem, the ranks of $\mathbf{O}$ and $\mathbf{C}$ can no longer increase after the N^{th} block-row or block-column, respectively.

In Appendix A.3, we show how to construct a minimal realization of a given transfer function.

A.2 Proof of Propositions 1 and 2

Proof. Applying Eq. (11), we may write the input-output maps for both algorithms as $\hat{y}(z) = \hat{O}_i(z)x_i^0 + \hat{H}_i(z)\hat{u}(z)$ for $i = 1, 2$. If there exist x_1^0 and x_2^0 such that the input-output maps of both systems are the same, we must have $\hat{O}_1(z)x_1^0 = \hat{O}_2(z)x_2^0$ and $\hat{H}_1(z) = \hat{H}_2(z)$. This establishes necessity for Propositions 1 and 2.

We now prove sufficiency. If $\hat{H}_1(z) = \hat{H}_2(z)$, both systems have the same input-output map if and only if $\hat{O}_1(z)x_1^0 = \hat{O}_2(z)x_2^0$. A trivial solution is $x_1^0 = 0$ and $x_2^0 = 0$, which proves sufficiency for Proposition 1. Now suppose both realizations are minimal and pick any x_2^0. By Proposition 12, there exists an invertible matrix T such that: $(A_2, B_2, C_2, D_2) = (TA_1T^{-1}, TB_1, C_1T^{-1}D_1)$. Therefore, we have:

$$\begin{aligned}\hat{O}_2(z)x_2^0 &= zC_2(zI - A_2)^{-1}x_2^0 = zC_1T^{-1}(zI - TA_1T^{-1})^{-1}x_2^0 \\ &= zC_1(zI - A_1)^{-1}T^{-1}x_2^0 = \hat{O}_1(z)T^{-1}x_2^0.\end{aligned}$$

Setting $x_1^0 = T^{-1}x_2^0$ leads us to $\hat{O}_1(z)x_1^0 = \hat{O}_2(z)x_2^0$, as required. To prove uniqueness, suppose $\tilde{x}_1^0 \neq x_1^0$ is a different solution, so that $\hat{O}_1(z)x_1^0 = \hat{O}_1(z)\tilde{x}_1^0$. In other words, $\hat{O}_1(z)v = 0$ for some $v := \tilde{x}_1^0 - x_1^0 \neq 0$. Then, we have:

$$\hat{O}_1(z)v = zC_1(zI - A_1)^{-1}v = \left(C_1 + C_1A_1z^{-1} + C_1A_1^2z^{-2} + \cdots\right)v = 0.$$

We conclude that $C_1A_1^kv = 0$ for $k = 0, 1, \ldots$, and therefore $\mathcal{O}_1 v = 0$, where $\mathcal{O}_1$ is the observability matrix (Definition 8). Minimality of (A_1, B_1, C_1, D_1) implies observability by Proposition 12, and therefore $\mathcal{O}_1$ has full column rank and $v = 0$, a contradiction. This establishes sufficiency of Proposition 2 and completes the proof. □

A.3 From transfer functions to algorithms

We begin by summarizing some key properties of transfer functions.

Proposition 14. *Consider a state-space system (A, B, C, D) and its associated transfer function $\hat{H}(z) = D + C(zI - A)^{-1}B$.*

1. *$\hat{H}$ is* rational*, which means each entry $\hat{H}_{ij}(z)$ can be expressed as a ratio of polynomials $\hat{H}_{ij}(z) = \frac{p_{ij}(z)}{q_{ij}(z)}$ where $p_{ij}(z)$ and $q_{ij}(z)$ have no common factors.*
2. *$\hat{H}$ is* proper*, which means that $\deg(p_{ij}) \leq \deg(q_{ij})$ for all i, j.*
3. *$\hat{H}$ is* strictly proper*, meaning $\deg(p_{ij}) < \deg(q_{ij})$ for all i, j, if and only if $D = 0$.*

Proposition 14 follows immediately from the formula $\hat{H}(z) = D + C(zI - A)^{-1}B$.

From transfer functions to minimal realizations

Given that all state-space realizations yield proper transfer functions, we now show how to construct a minimal realization given an arbitrary $p \times m$ proper rational transfer function $\hat{H}(z)$.[6] There are many efficient methods for constructing realizations from transfer functions [29, §5.4]. We now present one such method, the Ho–Kalman algorithm [60], which is direct, efficient, and numerically stable.

The Ho–Kalman algorithm is based on the Hankel matrix and leverages Proposition 13 and Remark 7. Let N be an upper bound on the number of states of the minimal realization. One way to obtain such a bound is to let N be the degree of the denominator polynomial of $\det \hat{H}(z)$. Now form the truncated Hankel matrix

$$\mathcal{H}_N := \begin{bmatrix} M_1 & M_2 & \cdots & M_N \\ M_2 & M_3 & \cdots & M_{N+1} \\ \vdots & \vdots & \ddots & \vdots \\ M_N & M_{N+1} & \cdots & M_{2N-1} \end{bmatrix} = \underbrace{\begin{bmatrix} C \\ CA \\ \vdots \\ CA^{N-1} \end{bmatrix}}_{\mathcal{O}_N} \underbrace{\begin{bmatrix} B & AB & \cdots & A^{N-1}B \end{bmatrix}}_{\mathcal{C}_N} = \mathcal{O}_N \mathcal{C}_N.$$

The *shifted* Hankel matrix (starts at M_2 instead of M_1) can also be factored:

$$\mathcal{H}_N^+ := \begin{bmatrix} M_2 & M_3 & \cdots & M_{N+1} \\ M_3 & M_4 & \cdots & M_{N+2} \\ \vdots & \vdots & \ddots & \vdots \\ M_{N+1} & M_{N+2} & \cdots & M_{2N} \end{bmatrix} = \mathcal{O}_N A \mathcal{C}_N.$$

Now compute the compact singular value decomposition (SVD) $\mathcal{H}_N = U\Sigma V^\top$. By Proposition 13, $\Sigma \in \mathbb{R}^{n \times n}$ and n is the minimal state dimension. We will choose $\mathcal{O}_N = U\Sigma^{1/2}$ and $\mathcal{C}_N = \Sigma^{1/2}V^\top$. These matrices are left- and right-invertible, respectively. Finally, define the following matrices.

- $D = M_0 = \lim_{z \to \infty} \hat{H}(z)$.

[6] When representing algorithms, we typically have $p = m$ (square $\hat{H}$), since there are as many oracle inputs as oracle outputs. In general, we can find state-space realizations for non-square transfer functions.

- C is the first p rows of $\mathcal{O}_N = U\Sigma^{1/2}$.
- B is the first m columns of $\mathcal{C}_N = \Sigma^{1/2}V^\top$.
- $A = \mathcal{O}_N^\dagger \mathcal{H}_N^+ \mathcal{C}_N^\dagger = \Sigma^{-1/2}U^\top \mathcal{H}_N^+ V\Sigma^{-1/2}$.

Then, (A, B, C, D) is a minimal realization of $\hat{H}(z)$. In practice, one typically chooses N to be a loose upper bound, such as twice the degree of the determinant $\det \hat{H}(z)$, as this yields a more numerically stable SVD computation.

Remark 8. In the field of controls, the Ho–Kalman algorithm is often used as a method of *system identification*, where the Markov parameters M_k are measured in a physical system, and we seek a state-space model (A, B, C, D) or transfer function model $\hat{H}(z)$ that fits the data [60]. Hankel singular values also show up in *model reduction*, when we seek simpler approximate models (with fewer states). One way is to truncate the smallest Hankel singular values [49, §7]. This is akin to finding a low-rank approximation of a real matrix by truncating the smallest singular values.

From realizations to state update equations

As described in Section 4, state-space realization can directly be converted back to step-by-step implementations of the form of Algorithm 4.1. These implementations will be explicit if D can be permuted into a strictly lower-triangular matrix, and implicit otherwise.

Appendix B Equivalence proofs

B.1 Proof of Lemma 3

We verify each property separately.

Reflexivity: Let $\hat{\Delta}_m = I$. For any $\hat{H}$, we trivially have $\hat{H} = \hat{\Delta}_m \hat{H} \hat{\Delta}_m^{-1}$. Therefore $\hat{H} \sim \hat{H}$, establishing reflexivity.

Symmetry: Suppose $\hat{H}_1 \sim \hat{H}_2$. Therefore,

$$\hat{H}_1 = \hat{\Delta}_m \hat{H}_2 \hat{\Delta}_m^{-1} \implies \hat{H}_2 = \hat{\Delta}_m^{-1} \hat{H}_1 \hat{\Delta}_m = \big(\hat{\Delta}_m^{-1} z^{-M}\big) \hat{H}_1 \big(\hat{\Delta}_m^{-1} z^{-M}\big)^{-1},$$

where we let $M := \max(m_1, \ldots, m_p)$. Therefore, $\hat{\Delta}_m^{-1} z^{-M}$ is a valid multi-shift (all powers of z are nonpositive) and $\hat{H}_2 \sim \hat{H}_1$, establishing symmetry.

Transitivity: Suppose $\hat{H}_1 \sim \hat{H}_2$ and $\hat{H}_2 \sim \hat{H}_3$. Therefore there exist multi-shifts $\hat{\Delta}_1$ and $\hat{\Delta}_2$ such that $\hat{H}_1 = \hat{\Delta}_1 \hat{H}_2 \hat{\Delta}_1^{-1}$ and $\hat{H}_2 = \hat{\Delta}_2 \hat{H}_3 \hat{\Delta}_2^{-1}$. Thus, we have

$$\hat{H}_1 = \hat{\Delta}_1 \hat{H}_2 \hat{\Delta}_1^{-1} = \hat{\Delta}_1 \hat{\Delta}_2 \hat{H}_3 \hat{\Delta}_2^{-1} \hat{\Delta}_1^{-1} = \big(\hat{\Delta}_1 \hat{\Delta}_2\big) \hat{H}_3 \big(\hat{\Delta}_1 \hat{\Delta}_2\big)^{-1}.$$

Since $\hat{\Delta}_1 \hat{\Delta}_2$ is a valid multi-shift, we have $\hat{H}_1 \sim \hat{H}_3$, establishing transitivity. □

B.2 Proof of Theorem 5

Suppose that $\mathcal{R}\begin{bmatrix}\hat{H}_1\\ I\end{bmatrix} = \hat{M}\mathcal{R}\begin{bmatrix}\hat{H}_2\\ I\end{bmatrix}$. Then for any $\hat{y}_2$, there exists a $\hat{y}_1$ such that $\begin{bmatrix}\hat{H}_1\hat{y}_1\\ \hat{y}_1\end{bmatrix} = \hat{M}\begin{bmatrix}\hat{H}_2\hat{y}_2\\ \hat{y}_2\end{bmatrix}$. Multiplying both sides on the left by $\begin{bmatrix}I & -\hat{H}_1\end{bmatrix}$, we obtain $\begin{bmatrix}I & -\hat{H}_1\end{bmatrix}\hat{M}\begin{bmatrix}\hat{H}_2\\ I\end{bmatrix}\hat{y}_2 = 0$. This holds for all $\hat{y}_2$, therefore $\begin{bmatrix}I & -\hat{H}_1\end{bmatrix}\hat{M}\begin{bmatrix}\hat{H}_2\\ I\end{bmatrix} = 0$.

Conversely, suppose that $\begin{bmatrix}I & -\hat{H}_1\end{bmatrix}\hat{M}\begin{bmatrix}\hat{H}_2\\ I\end{bmatrix} = 0$. Augment the block matrices to obtain: $\begin{bmatrix}I & -\hat{H}_1\\ 0 & I\end{bmatrix}\hat{M}\begin{bmatrix}I & \hat{H}_2\\ 0 & I\end{bmatrix} = \begin{bmatrix}\star & 0\\ \star & \star\end{bmatrix}$, where the $\star$'s denote unimportant blocks. The left-hand side is invertible, so the right-hand side is invertible as well. Inverting both sides, the right-hand side remains block-lower triangular, and we obtain $\begin{bmatrix}I & -\hat{H}_2\\ 0 & I\end{bmatrix}\hat{M}^{-1}\begin{bmatrix}I & \hat{H}_1\\ 0 & I\end{bmatrix} = \begin{bmatrix}\star & 0\\ \star & \star\end{bmatrix}$, where the $\star$'s indicate different blocks from before. Extracting the $(1,2)$ block, we obtain $\begin{bmatrix}I & -\hat{H}_2\end{bmatrix}\hat{M}^{-1}\begin{bmatrix}\hat{H}_1\\ I\end{bmatrix} = 0$.

Now, pick $\begin{bmatrix}\hat{u}\\ \hat{v}\end{bmatrix} \in \hat{M}\mathcal{R}\begin{bmatrix}\hat{H}_2\\ I\end{bmatrix}$, so there exists some $\hat{y}$ such that $\begin{bmatrix}\hat{u}\\ \hat{v}\end{bmatrix} = \hat{M}\begin{bmatrix}\hat{H}_2\\ I\end{bmatrix}\hat{y}$. Multiplying on the left by $\begin{bmatrix}I & -\hat{H}_1\end{bmatrix}$, we conclude that $\hat{u} = \hat{H}_1\hat{v}$, which we can rewrite as $\begin{bmatrix}\hat{u}\\ \hat{v}\end{bmatrix} = \begin{bmatrix}\hat{H}_1\\ I\end{bmatrix}\hat{v} \in \mathcal{R}\begin{bmatrix}\hat{H}_1\\ I\end{bmatrix}$. Therefore, $\mathcal{R}\begin{bmatrix}\hat{H}_1\\ I\end{bmatrix} \supseteq \hat{M}\mathcal{R}\begin{bmatrix}\hat{H}_2\\ I\end{bmatrix}$. Similarly, pick $\begin{bmatrix}\hat{u}\\ \hat{v}\end{bmatrix} \in \hat{M}^{-1}\mathcal{R}\begin{bmatrix}\hat{H}_1\\ I\end{bmatrix}$. Following similar steps, we obtain $\mathcal{R}\begin{bmatrix}\hat{H}_2\\ I\end{bmatrix} \supseteq \hat{M}^{-1}\mathcal{R}\begin{bmatrix}\hat{H}_1\\ I\end{bmatrix}$. Combining the two inclusions above, we obtain $\mathcal{R}\begin{bmatrix}\hat{H}_1\\ I\end{bmatrix} = \hat{M}\mathcal{R}\begin{bmatrix}\hat{H}_2\\ I\end{bmatrix}$, as required. □

References

[1] Popov, L.D.: A modification of the arrow-hurwicz method for search of saddle points. Mathematical notes of the Academy of Sciences of the USSR **28**(5), 845–848 (1980)

[2] Chiang, C.-K., Yang, T., Lee, C.-J., Mahdavi, M., Lu, C.-J., Jin, R., Zhu, S.: Online optimization with gradual variations. In: Conference on Learning Theory, pp. 1–6 (2012)

[3] Gidel, G., Berard, H., Vignoud, G., Vincent, P., Lacoste-Julien, S.: A variational inequality perspective on generative adversarial networks. In: International Conference on Learning Representations (2019)

[4] Rakhlin, A., Sridharan, K.: Online learning with predictable sequences. Proceedings of Machine Learning Research, vol. 30, pp. 993–1019 (2013)

[5] Daskalakis, C., Ilyas, A., Syrgkanis, V., Zeng, H.: Training GANs with optimism. In: International Conference on Learning Representations (2018)

[6] Malitsky, Y.: Projected reflected gradient methods for monotone variational inequalities. SIAM Journal on Optimization **25**(1), 502–520 (2015)

[7] Grant, M., Boyd, S.: CVX: Matlab Software for Disciplined Convex Programming, version 2.1. http://cvxr.com/cvx (2014)

[8] Grant, M., Boyd, S.: Graph implementations for nonsmooth convex programs. In: Recent Advances in Learning and Control. Lecture Notes in Control and Information Sciences, pp. 95–110. Springer, London (2008)

[9] Udell, M., Mohan, K., Zeng, D., Hong, J., Diamond, S., Boyd, S.: Convex optimization in julia. In: 2014 First Workshop for High Performance Technical Computing in Dynamic Languages, pp. 18–28 (2014). IEEE

[10] Diamond, S., Boyd, S.: Cvxpy: A python-embedded modeling language for convex optimization. The Journal of Machine Learning Research **17**(1), 2909–2913 (2016)

[11] Shen, X., Diamond, S., Udell, M., Gu, Y., Boyd, S.: Disciplined multi-convex programming. In: 2017 29th Chinese Control And Decision Conference (CCDC), pp. 895–900 (2017). IEEE

[12] Lessard, L., Recht, B., Packard, A.: Analysis and design of optimization algorithms via integral quadratic constraints. SIAM Journal on Optimization **26**(1), 57–95 (2016)

[13] Ryu, E.K., Yin, W.: Large-scale Convex Optimization: Algorithms & Analyses Via Monotone Operators. Cambridge University Press, Cambridge, United Kingdom (2022)

[14] Boyd, S., Parikh, N., Chu, E., Peleato, B., Eckstein, J.: Distributed optimization and statistical learning via the alternating direction method of multipliers. Foundations and Trends in Machine Learning **3**, 1–122 (2011)

[15] Fukushima, M.: Application of the alternating direction method of multipliers to separable convex programming problems. Computational Optimization and Applications **1**, 93–111 (1992)

[16] Eckstein, J., Fukushima, M.: Some reformulations and applications of the alternating direction method of multipliers. Large Scale Optimization: State of the Art, 119–138 (1993)

[17] Eckstein, J.: Splitting methods for monotone operators with applications to parallel optimization. PhD thesis, MIT (1989)

[18] Chambolle, A., Pock, T.: A first-order primal-dual algorithm for convex problems with applications to imaging. Journal of mathematical imaging and vision **40**, 120–145 (2011)

[19] Douglas, J., Rachford, H.H.: On the numerical solution of heat conduction problems in two and three space variables. Transactions of the American mathematical Society **82**, 421–439 (1956)

[20] Eckstein, J., Bertsekas, D.P.: On the Douglas-Rachford splitting method and the proximal point algorithm for maximal monotone operators. Mathematical Programming **55**, 293–318 (1992)

[21] Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge University Press, Cambridge, United Kingdom (2004)

[22] Bubeck, S.: Convex optimization: algorithms and complexity. Foundations and Trends® in Machine Learning **8**(3-4), 231–357 (2015)

[23] Nesterov, Y.: Lectures on Convex Optimization. Springer, Berlin (2018)

[24] Beck, A.: First-order Methods in Optimization. SIAM, Philadelphia, PA (2017)

[25] Fenchel, W.: Convex Cones, Sets and Functions, Mimeographed Notes. Princeton University Press, Princeton, NJ (1953)

[26] Parikh, N., Boyd, S.: Proximal algorithms. Foundations and Trends in Optimization **1**, 127–239 (2014)

[27] Hu, B., Seiler, P., Lessard, L.: Analysis of biased stochastic gradient descent using sequential semidefinite programs. Mathematical Programming, 1–26 (2020)

[28] Williams, R.L., Lawrence, D.A.: Linear State-space Control Systems. John Wiley & Sons, Hoboken, NJ (2007)

[29] Antsaklis, P.J., Michel, A.N.: Linear Systems. Birkhäuser, Boston, MA (2006)

[30] Polyak, B.T.: Some methods of speeding up the convergence of iteration methods. USSR computational mathematics and mathematical physics **4**(5), 1–17 (1964)

[31] Nesterov, Y.: A method for solving the convex programming problem with convergence rate o (1/k2). In: Dokl Akad Nauk Sssr, vol. 269, p. 543 (1983)

[32] Van Scoy, B., Freeman, R.A., Lynch, K.M.: The fastest known globally convergent first-order method for minimizing strongly convex functions. IEEE Control Systems Letters **2**(1), 49–54 (2018)

[33] Ma, J., Yarats, D.: Quasi-hyperbolic momentum and adam for deep learning. In: International Conference on Learning Representations (2019)

[34] Yan, Y., Yang, T., Li, Z., Lin, Q., Yang, Y.: A unified analysis of stochastic momentum methods for deep learning. In: IJCAI International Joint Conference on Artificial Intelligence (2018)

[35] Shen, L., Chen, C., Zou, F., Jie, Z., Sun, J., Liu, W.: A unified analysis of adagrad with weighted aggregation and momentum acceleration. IEEE Transactions on Neural Networks and Learning Systems (2023)

[36] Nedic, A., Ozdaglar, A.: Distributed subgradient methods for multi-agent optimization. IEEE Transactions on Automatic Control **54**(1), 48–61 (2009)

[37] Shi, W., Ling, Q., Wu, G., Yin, W.: EXTRA: An exact first-order algorithm for decentralized consensus optimization. SIAM Journal on Optimization **25**(2), 944–966 (2015)

[38] Li, Z., Shi, W., Yan, M.: A decentralized proximal-gradient method with network independent step-sizes and separated convergence rates. IEEE Transactions on Signal Processing **67**(17), 4494–4506 (2019)

[39] Yuan, K., Ying, B., Zhao, X., Sayed, A.H.: Exact diffusion for distributed optimization and learning—Part I: Algorithm development. IEEE Transactions on Signal Processing **67**(3), 708–723 (2018)

[40] Sundararajan, A., Scoy, B.V., Lessard, L.: A canonical form for first-order distributed optimization algorithms. In: American Control Conference, pp. 4075–4080 (2019)

[41] Condat, L.: A primal–dual splitting method for convex optimization involving lipschitzian, proximable and linear composite terms. Journal of optimization theory and applications **158**(2), 460–479 (2013)

[42] Vũ, B.C.: A splitting algorithm for dual monotone inclusions involving cocoercive operators. Advances in Computational Mathematics **38**, 667–681 (2013)

[43] Yan, M.: A new primal–dual algorithm for minimizing the sum of three functions with a linear operator. Journal of Scientific Computing **76**, 1698–1717 (2018)

[44] Salim, A., Condat, L., Mishchenko, K., Richtárik, P.: Dualize, split, randomize: Toward fast nonsmooth optimization algorithms. Journal of Optimization Theory and Applications **195**(1), 102–130 (2022)

[45] Jiang, X., Vandenberghe, L.: Bregman three-operator splitting methods. Journal of Optimization Theory and Applications **196**(3), 936–972 (2023)

[46] Ryu, E.K., Boyd, S.: Primer on monotone operator methods. Appl. Comput. Math **15**(1), 3–43 (2016)

[47] Beck, A., Teboulle, M.: A fast iterative shrinkage-thresholding algorithm for linear

inverse problems. SIAM Journal on Imaging Sciences **2**, 183–202 (2009)

[48] Moreau, J.J.: Décomposition orthogonale d'un espace hilbertien selon deux cônes mutuellement polaires. Comptes rendus hebdomadaires des séances de l'Académie des sciences **255**, 238–240 (1962)

[49] Zhou, K., Doyle, J.C., Glover, K.: Robust and Optimal Control. Prentice-Hall, Englewood Cliffs, NJ (1996)

[50] O'Connor, D., Vandenberghe, L.: On the equivalence of the primal-dual hybrid gradient method and douglas–rachford splitting. Mathematical Programming **179**, 85–108 (2020)

[51] Davis, D., Yin, W.: A three-operator splitting scheme and its optimization applications. Set-valued and variational analysis **25**, 829–858 (2017)

[52] Vasilyev, F., Khoroshilova, E., Antipin, A.: An extragradient method for finding the saddle point in an optimal control problem. Moscow University Computational Mathematics and Cybernetics **34**(3), 113–118 (2010)

[53] Censor, Y., Gibali, A., Reich, S.: The subgradient extragradient method for solving variational inequalities in hilbert space. Journal of Optimization Theory and Applications **148**(2), 318–335 (2011)

[54] Kamen, E.W., Khargonekar, P.P., Poolla, K.R.: A transfer-function approach to linear time-varying discrete-time systems. SIAM Journal on Control and Optimization **23**(4), 550–565 (1985) https://doi.org/10.1137/0323035

[55] Mohammadpour, J., Scherer, C.W.: Control of Linear Parameter Varying Systems with Applications. Springer, New York (2012)

[56] Sun, Z.: Switched Linear Systems: Control and Design. Springer, London, United Kingdom (2006)

[57] Yan, M., Yin, W.: Self equivalence of the alternating direction method of multipliers. Splitting Methods in Communication, Imaging, Science, and Engineering, 165–194 (2016)

[58] Zhao, S., Lessard, L., Udell, M.: An automatic system to detect equivalence between iterative algorithms (2022). https://arxiv.org/abs/2105.04684

[59] Michalowsky, S., Scherer, C., Ebenbauer, C.: Robust and structure exploiting optimisation algorithms: an integral quadratic constraint approach. International Journal of Control **94**(11), 2956–2979 (2021)

[60] Ho, B., Kálmán, R.E.: Effective construction of linear state-variable models from input/output functions. at-Automatisierungstechnik **14**(1-12), 545–548 (1966)