AN AUTOMATIC SYSTEM TO DETECT EQUIVALENCE BETWEEN ITERATIVE ALGORITHMS

SHIPU ZHAO*, LAURENT LESSARD†, AND MADELEINE UDELL*

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 some algorithm transformations including permutations of the update equations, repetition of the iteration, and conjugation of some of the function oracles in the algorithm. To support the paper, we have developed a software package named LINNAEUS that implements the framework to identify other iterative algorithms that are equivalent to an input algorithm. More broadly, this framework and software advances the goal of making mathematics searchable.

Key words. 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 in order 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 [26] in the context of solving saddle point problems. This method was later generalized by Chiang et al. [9, §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) [14]. Algorithm 1.3 is an adaptation of *Optimistic Mirror Descent* [27] used by Daskalakis et al. [10] and also used to train GANs. Finally, algorithm 1.4 was proposed by Malitsky [20] in the context of 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 $F^k(\cdot)$ is the gradient of the loss function at time step k.

Algorithm 1.1 (Modified Arrow–Hurwicz)

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

Algorithm 1.2 (Extrapolation from the past)

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

Algorithm 1.3 (Optimistic Mirror Descent)

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

Algorithm 1.4 (Reflected Gradient Method)

for $k = 1, 2, \dots$ **do**
$x_1^{k+1} = x_1^k - \eta F^k(2x_1^k - x_1^{k-1})$
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 [14], algorithm equivalence is not always immediately apparent. Indeed, it is not uncommon for algorithms to be unknowingly re-discovered in a different but equivalent form and given a new name before anyone observes that they are not new after all.

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:

*Cornell University (sz533@cornell.edu, udell@cornell.edu).
†Northeastern University (l.lessard@northeastern.edu).

[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.

- A universal way of representing algorithms, inspired by the literature on control theory. Specifically, we will use *state-space realizations* and *transfer functions*.
- A description of different ways in which two algorithms can be deemed equivalent, and how these equivalences manifest themselves with regards to the algorithm representation.
- A computationally efficient way of verifying whether two algorithms belong to the same equivalence class and how to transform between equivalent representations.

We also present a software package we named LINNAEUS[2], for the classification and taxonomy of iterative algorithms. The software is a search engine, where the input is an algorithm described using natural syntax, and the output is a canonical form for the algorithm along with any known names and pointers to relevant literature. The approach described in this paper allows LINNAEUS to search over first order optimization algorithms such as gradient descent with acceleration, the alternating directions method of multipliers (ADMM), and the extragradient method. As the database in LINNAEUS grows, it will help algorithm researchers understand and efficiently discover connections between algorithms. More generally, LINNAEUS advances the goal of making mathematics searchable.

This paper is organized as follows. In section 2, we introduce three examples of equivalent algorithms that motivate our framework. In section 3, we briefly review important background on linear systems and optimization used throughout the paper. We formally define two notions of algorithm equivalence, *oracle equivalence* and *shift equivalence*, in section 4 and discuss how to characterize them via transfer functions in sections 5 and 6. Certain transformations can also be identified and characterized with our framework including *algorithm repetition*, repeating an algorithm multiple times, and *conjugation*, a transformation using conjugate function oracles. These are discussed in sections 7 and 8 respectively. In section 9, we introduce our software package LINNAEUS for the classification of iterative algorithms.

2. Motivating examples.

To explain what we mean by algorithm equivalence, we introduce three motivating examples in this section. Each provides a different view of how two algorithms might be equivalent.

Algorithm 2.1

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

Algorithm 2.2

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

The first example consists of algorithms 2.1 and 2.2. These algorithms are equivalent in a strong sense: when suitably initialized, we may transform the iterates of algorithm 2.1 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 2.2. We say that the sequences $(x_1^k)_{k\geq 0}$ and $(x_2^k)_{k\geq 0}$ are *equivalent* to sequences $(\xi_1^k)_{k\geq 0}$ and $(\xi_2^k)_{k\geq 0}$ *up to an invertible linear transformation.*

Algorithm 2.3

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

Algorithm 2.4

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 2.3 and 2.4. 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 2.3 by the linear map $\xi^k = -x_1^k + 2x_2^k$ to yield the iterates of algorithm 2.4. 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.

[2]Named after Carl Linnaeus, a botanist and zoologist who invented the modern system of naming organisms.

Algorithm 2.5

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

Algorithm 2.6

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

The third example consists of algorithms 2.5 and 2.6. With suitable initialization, they will generate the same sequence of calls to the proximal operator, ignoring the very first call to one of the oracles. Specifically, algorithm 2.6 is initialized as $\xi_1^0 = x_3^0$, $\xi_2^0 = x_1^1$ and the first call to prox_f in algorithm 2.5 is ignored. We will say they are equivalent up to a prefix or shift: they satisfy *shift equivalence*.

Generalizing from these motivating examples, we will call algorithms equivalent when they generate an identical sequence (e.g., of states or oracle calls) up to some transformations, with suitable initialization. To make our ideas formal, we need a few definitions and some ideas from control theory. We will then revisit those motivating examples and define algorithm equivalence.

3. Preliminaries. We let $\mathbb{R}^n$ denote the standard Euclidean space of n-dimensional vectors, and use boldface lowercase symbols denote semi-infinite sequences of vectors, which we index using superscripts. For example, we may write $\mathbf{x} := (x^0, x^1, \dots)$, where $x^k \in \mathbb{R}^n$ for each $k \geq 0$. Subscripts index components or subvectors: for example, we may write $x = \left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right] \in \mathbb{R}^n$, where $x_1 \in \mathbb{R}^{n_1}$ and $x_2 \in \mathbb{R}^{n-n_1}$.

3.1. Optimization.

Optimization problem, objective, and constraints. An optimization problem is identified by an objective function and a constraint set. The objective may be written as the sum of several functions, and the constraint set may be the intersection of several sets. As an example, in the optimization problem (3.1) [4]

$$\begin{array}{ll} \text{minimize} & f(x) + g(z) \\ \text{subject to} & Ax + Bz = c, \end{array} \tag{3.1}$$

the objective function is $f(x) + g(z)$ and the constraint set is $\{(x, z) : Ax + Bz = c\}$.

Oracles. We assume an oracle model of optimization: we can only access an optimization problem by querying oracles at discrete query points [5, §4; 6, §1; 23, §1]. Oracles might include the gradient or proximal operator of a function, or projection onto a constraint set [2, §6; 13, §2; 25, §1]. Each query to the oracle returns an output such as the function value, gradient, or proximal operator. For example, the oracles for problem (3.1) might include the gradients or proximal operators of f and g, and projection onto the hyperplane $\{(x, z) : Ax + Bz = c\}$.

3.2. Algorithms.

Detecting equivalence between *any* pair of algorithms is beyond the scope of this paper. Instead, we restrict our attention to equivalence between iterative linear time invariant optimization algorithms. In the following section, we provide some intuition and define each of these terms. Further formalism of these terms will be provided in the next subsection on control theory.

Iterative algorithms. Given an optimization problem and an initial point $x^0 \in \mathcal{X}$, an *iterative algorithm* $\mathcal{A}$ generates a sequence of points $\mathbf{x} := (x^k)_{k \geq 0}$ by repeated application of the map $\mathcal{A} : \mathcal{X} \to \mathcal{X}$. (We do not distinguish the algorithm from its associated map.) Hence, $x^{k+1} = \mathcal{A}(x^k)$ for $k \geq 0$. We call x^k the *state* of the algorithm at *time* k. We make two important simplifying assumptions when treating algorithms.

First, suppose the operator $\mathcal{A}$ calls each different oracle *exactly once*. (We will see how to extend our ideas to more complex algorithms later.) This assumption forbids trivial repetition, such as $\mathcal{A}' := \mathcal{A} \circ \mathcal{A}$. Second, we consider algorithms that are *time-invariant*. In general, one could envision an algorithm $\mathcal{A}^k$ that changes at each timestep. Such time-varying algorithms are common in practice: for example, gradient-based methods with diminishing stepsizes. We view time-varying algorithms as a scheme for switching between different time-invariant algorithms. Since our aim is to reason about algorithm equivalence, we restrict our attention to time-invariant algorithms. A nice benefit of only this restriction is that we can define algorithm equivalence independently of the choice of initial point.

The formulation $x^{k+1} = \mathcal{A}(x^k)$ is general enough to include algorithms with multiple timesteps. For example consider algorithm 1.4: $x_1^{k+1} = x_1^k - \eta F(2x_1^k - x_1^{k-1})$. If we define the new state $x_2^k := x_1^{k-1}$ and let

$x^k := \begin{bmatrix} x_1^k \\ x_2^k \end{bmatrix}$, then we may rewrite the algorithm as

$$x^{k+1} = \begin{bmatrix} x_1^{k+1} \\ x_2^{k+1} \end{bmatrix} = \begin{bmatrix} x_1^k - \eta F(2x_1^k - x_2^k) \\ x_1^k \end{bmatrix} = \mathcal{A}\left(\begin{bmatrix} x_1^k \\ x_2^k \end{bmatrix} \right) = \mathcal{A}(x^k). \tag{3.2}$$

The algorithm $\mathcal{A}$ contains a combination of oracle calls and state updates. Define y^k and u^k to be the input and output of the oracles called at time k, respectively. Now, write three separate equations for the state update, oracle input, and oracle output. Applying this to (3.2), we obtain:

$$\begin{bmatrix} x_1^{k+1} \\ x_2^{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1^k \\ x_2^k \end{bmatrix} + \begin{bmatrix} -\eta \\ 0 \end{bmatrix} u^k \qquad \text{(state update)}, \tag{3.3a}$$

$$y^k = \begin{bmatrix} 2 & -1 \end{bmatrix} \begin{bmatrix} x_1^k \\ x_2^k \end{bmatrix} \qquad \text{(oracle input)}, \tag{3.3b}$$

$$u^k = F(y^k) \qquad \text{(oracle output)}. \tag{3.3c}$$

Oracle sequence. We have defined an algorithm $\mathcal{A}$ as a map $\mathcal{X} \to \mathcal{X}$. In optimization, it is also conventional to write an algorithm as a sequence of update equations, that are executed sequentially on a computer to implement the map. When this sequence of updates is executed, we may record the sequence of states or the sequence of oracle calls (oracle and input pairs), which we call the oracle sequence. There may be several ways of writing the algorithm as a sequence of updates, which may produce different state sequences or oracle sequences. We are not aware of any practical algorithm for optimization that may be written to produce two different oracle sequences. Hence we will assume for now that the oracle sequence produced by an algorithm is unique. We will revisit this assumption later in the paper (section 6) to see how our ideas extend to more complex (not-yet-discovered) algorithms.

Linear algorithms. The equations (3.3) have the general *linear* form

$$x^{k+1} = Ax^k + Bu^k, \tag{3.4a}$$

$$y^k = Cx^k + Du^k, \tag{3.4b}$$

$$u^k = \phi(y^k). \tag{3.4c}$$

We say that a time-invariant algorithm is *linear* if it can be written in the form of (3.4), where x^k is the algorithm state and ϕ is the set of oracles.

In the rest of the paper, unless specifically noted, our discussion is limited to linear algorithms. We will see that linear algorithms are a rich class that includes many commonly used algorithms, such many accelerated methods, proximal methods, operator splitting methods, and more [16, 18].

The general form (3.4) represents a convenient parameterization of linear algorithms in terms of matrices (A, B, C, D), but it is only a starting point. For example, algorithms 1.1–1.4 have different (A, B, C, D) parameters despite being equivalent algorithms. In the next section, we show how tools from control theory can be brought to bear on these sorts of representations.

3.3. Control theory. This subsection provides a brief overview of relevant methods and terminology from control theory. More detail can be found in standard references such as [1, Ch. 1–3] and [35, Ch. 1,2,5].

Algorithms as linear systems. Let $\mathbf{u}$ denote the entire sequence of u^k and $\mathbf{y}$ denote the entire sequence of y^k. The equations in (3.4) can be separated into two parts. Equations (3.4a) and (3.4b) define a map $\mathbf{H}$ from $\mathbf{u}$ to $\mathbf{y}$ compactly as $\mathbf{y} = \mathbf{H}\mathbf{u}$, while (3.4c) defines a map $\mathbf{\Phi}$ from $\mathbf{y}$ to $\mathbf{u}$ as $\mathbf{u} = \mathbf{\Phi}\mathbf{y}$, where $\mathbf{\Phi} = \mathrm{diag}\{\phi, \phi, \dots\}$. We can represent these algebraic relations visually via the *block-diagram* shown in figure 1.

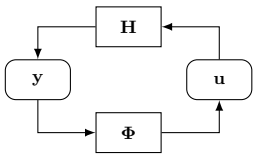

FIG. 1. *Block-diagram representation of an algorithm. This is equivalent to the pair of equations* $\mathbf{y} = \mathbf{H}\mathbf{u}$ *and* $\mathbf{u} = \mathbf{\Phi}\mathbf{y}$.

Consider map $\mathbf{H}$ defined by (3.4a) and (3.4b). For simplicity, we assume that $x^0 = 0$. As we eliminate $\{x^1, \dots, x^k\}$ from (3.4a) and (3.4b), map $\mathbf{H}$ can be represented as an semi-infinite matrix,

$$\begin{bmatrix} y^0 \\ y^1 \\ y^2 \\ y^3 \\ \vdots \end{bmatrix} = \underbrace{\begin{bmatrix} D & 0 & 0 & 0 & \cdots \\ CB & D & 0 & 0 & \cdots \\ CAB & CB & D & 0 & \cdots \\ C(A)^2B & CAB & CB & D & \cdots \\ \vdots & \ddots & \ddots & \ddots & \ddots \end{bmatrix}}_{\mathbf{H}} \begin{bmatrix} u^0 \\ u^1 \\ u^2 \\ u^3 \\ \vdots \end{bmatrix}. \tag{3.5}$$

In control theory, map $\mathbf{H}$ is considered as a (discrete-time) *system* that maps a sequence of *inputs* $\mathbf{u}$ to a sequence of *outputs* $\mathbf{y}$. Map $\mathbf{H}$ is linear since it can be represented as a semi-infinite matrix. The matrix representation is lower-triangular and it indicates $\mathbf{H}$ is *causal*. Further, $\mathbf{H}$ is time-invariant because the matrix representation is (block) *Toeplitz*, which means that $\mathbf{H}$ is (block) constant along diagonals from top-left to bottom right. Thus, $\mathbf{H}$ is a *causal linear time-invariant system*. For the rest of this paper, we will work with such systems and we will refer to such systems as *linear systems*.

Further, to combine maps $\mathbf{H}$ and $\mathbf{\Phi}$ together, a linear algorithm in the form of (3.4) can be regarded as a linear system connected in feedback with a nonlinearity shown by figure 1. At time k, u^k is the input and y^k is the output of the system. Nonlinear feedback ϕ represents the set of oracles such as the gradient or subgradient of a convex function and it maps the output y^k to the input u^k.

State-space realization. Reconsider equations (3.4a) and (3.4b). They correspond to the *state-space realization* of system $\mathbf{H}$. In control theory, a state-space realization is characterized by an internal sequence of *states* $\mathbf{x}$ that evolves according to a difference equation with parameters (A, B, C, D):

$$\begin{aligned} x^{k+1} &= Ax^k + Bu^k, \\ y^k &= Cx^k + Du^k, \end{aligned} \quad \text{or equivalently,} \quad \begin{bmatrix} x^{k+1} \\ y^k \end{bmatrix} = L \begin{bmatrix} x^k \\ u^k \end{bmatrix}, \text{ where } L = \begin{bmatrix} A & B \\ C & D \end{bmatrix}. \tag{3.6}$$

Here, $u^k \in \mathbb{R}^m$, $y^k \in \mathbb{R}^p$, and $x^k \in \mathbb{R}^n$. The parameters (A, B, C, D) are matrices of compatible dimensions, so $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, $C \in \mathbb{R}^{p\times n}$, and $D \in \mathbb{R}^{p\times m}$. The state-space realization corresponding to the system $\mathbf{H}$ can also be characterized by omitting all vectors and writing the block matrix L shown in (3.6) (right), which is the map from (x^k, u^k) to (x^{k+1}, y^k).

In this paper, we rely on such formalism that represents algorithms as linear systems using a state-space realization as (3.6) for each algorithm, following [16, 18]. The state-space realization L represents the linear part of an algorithm and map ϕ represents the nonlinear part. Moreover, we have $\mathcal{A} = (L, \phi)$. In this way, we can unroll figure 1 in time to obtain the block-diagram shown in figure 2. Each dashed box in figure 2 represents map $\mathcal{A}$ for each iteration.

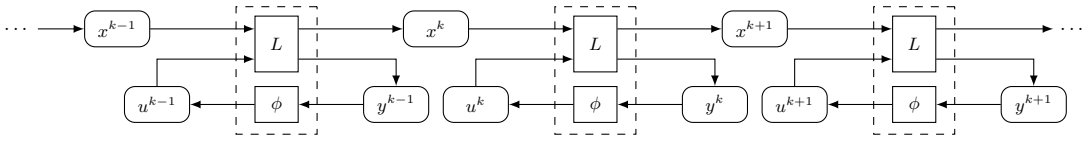

FIG. 2. *Unrolled-in-time block-diagram representation of an algorithm.*

Impulse response and transfer function. From (3.5), without the assumption that $x^0 = 0$, we can obtain

$$y^k = C(A)^k x^0 + \sum_{j=0}^{k-1} C(A)^{k-(j+1)} B u^j + Du^k. \tag{3.7}$$

The output y^k is the sum of $C(A)^k x^0$, which is due to the initial condition x^0, and $\sum_{j=0}^{k-1} C(A)^{k-(j+1)} B u^j + Du^k$, which is due to the inputs $\{u^0, \dots, u^k\}$. The compact form $\mathbf{y} = \mathbf{H}\mathbf{u}$ and its matrix representation (3.5) omit the first term that depends on x^0. These representations are formally equivalent to the state-space

model only when the state is initialized at $x_0 = 0$. However, linearity of $\mathbf{H}$ allows the two contributions to be studied separately:

$$(\text{total response}) = \underbrace{(\text{zero input response})}_{\text{set } u^k = 0 \text{ for } k \geq 0} + \underbrace{(\text{zero state response})}_{\text{set } x^k = 0}.$$

This decomposition is analogous to writing the general solution to a linear differential (or difference) equation as the sum of a homogeneous solution (due to initial conditions only) and a particular solution (due to the non-homogeneous terms only). We will characterize linear systems by their input-output map. The input-output map depends only on the zero state response, which allows us to avoid details about initialization. For simplicity, we denote the entries in the matrix representation of $\mathbf{H}$ in (3.5) as

$$H^k = \begin{cases} D & k = 0 \\ C(A)^{k-1}B & k \geq 1 \end{cases}. \tag{3.8}$$

To study the zero state response, recall from (3.5) that

$$y^k = H^k u^0 + H^{k-1}u^1 + \cdots + H^1 u^{k-1} + H^0 u^k. \tag{3.9}$$

The sequence $(H^k)_{k\geq 0}$ is called the *impulse response* of $\mathbf{H}$, because it corresponds to the impulsive input $u^0 = 1$ and $u^j = 0$ for $j \geq 1$.

A convenient way to represent $\mathbf{H}$ is via the use of a *transfer function*. To this end, we can represent $\mathbf{y}$ and $\mathbf{u}$ as generating functions in the variable z^{-1}. Equating powers of z^{-1}, we have:

$$\underbrace{\left(y^0 + y^1 z^{-1} + y^2 z^{-2} + \cdots\right)}_{\hat{y}(z)} = \underbrace{\left(H^0 + H^1 z^{-1} + H^2 z^{-2} + \cdots\right)}_{\hat{H}(z)} \underbrace{\left(u^0 + u^1 z^{-1} + u^2 z^{-2} + \cdots\right)}_{\hat{u}(z)}. \tag{3.10}$$

We can recover (3.9) by expanding the multiplication in (3.10) and grouping terms with the same power of z^{-1}. So when written as generating functions, the output is related to the input via multiplication. The functions $\hat{y}$ and $\hat{u}$ are the z-transforms of the sequences $\mathbf{y}$ and $\mathbf{u}$, respectively, and $\hat{H}$ is called the *transfer function*. If $p \geq 2$ or $m \geq 2$ (the H^k are matrices), then $\hat{H}$ is called the *transfer matrix*.

Substituting (3.8) into the definition of the transfer function, we can write a compact form for the formal power series $\hat{H}$, which converges on some appropriate set:

$$\hat{H}(z) = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] = D + \sum_{k=1}^{\infty} C(A)^{k-1}Bz^{-k} = C(zI - A)^{-1}B + D. \tag{3.11}$$

The transfer function $\hat{H}(z) = C(zI - A)^{-1}B + D$ can be directly computed from the state-space matrices (A, B, C, D). Moreover, $\hat{H}(z)$ is a matrix whose entries are rational functions of z. Hence the transfer function provides an computationally efficient way to uniquely characterize the input-output map of a system. We will use the block notation with solid lines to indicate transfer function as in (3.11).

Linear transformations of state-space realizations. Consider a linear transformation of the states x^k in (3.6). Specifically, suppose $Q \in \mathbb{R}^{n\times n}$ is invertible, and define $\tilde{x}^k = Qx^k$ for each k. The new state-space realization in terms of the new variables $\tilde{x}^k$ is

$$\begin{aligned} \tilde{x}^{k+1} &= QAQ^{-1}\tilde{x}^k + QBu^k \\ y^k &= CQ^{-1}\tilde{x}^k + Du^k \end{aligned}, \qquad \tilde{L} = \begin{bmatrix} QAQ^{-1} & QB \\ CQ^{-1} & D \end{bmatrix}. \tag{3.12}$$

It is straightforward to check that $\mathbf{H}$ and $\tilde{\mathbf{H}}$ have the same transfer function. Therefore, whether we apply the linear system $\mathbf{H}$ or $\tilde{\mathbf{H}}$, the same input sequence $\mathbf{u}$ will produce the same output sequence $\mathbf{y}$, although the respective states x^k and $\tilde{x}^k$ will generally be different. So although the state-space realization (A, B, C, D) depend on the coordinates used to represent states x^k, the transfer function is invariant under linear transformations.

This invariance is the key to understanding when two optimization algorithms are the same, even if they look different as written. For example, this idea alone suffices to show that algorithms 2.1 and 2.2 are equivalent.

Minimal realizations. Every set of appropriately-sized state-space parameters (A, B, C, D) produces a transfer matrix whose entries are rational functions of z. Closer inspection of the formula $\hat{H}(z) = C(zI - A)^{-1}B + D$ reveals that $\hat{H}(z) \to D$ as $z \to \infty$. Therefore, the rational entries of $\hat{H}(z)$ must be *proper*: the degree of the numerator cannot exceed the degree of the denominator. Moreover, the degree of the common denominator of all entries of $\hat{H}(z)$ cannot exceed n (the size of the matrix A).

The converse is also true: given any transfer matrix $\hat{H}(z)$ whose entries are proper with common denominator degree n, there exists a realization (A, B, C, D) where A has size at most n whose transfer function is $\hat{H}(z)$. Any realization of $\hat{H}(z)$ for which the size of A is as small as possible is called *minimal*. All minimal realizations of $\hat{H}(z)$ are related by an invertible state transformation via a suitably chosen invertible matrix Q, as in (3.12).

Realizations can be non-minimal when the transfer function has factors that cancel from both the numerator and denominator. For example, the following pair of state-space equations both have the same transfer function:

$$\hat{H}(z) = \left[\begin{array}{c|c} 1 & 1 \\ \hline 1 & 0 \end{array}\right] = 1 \cdot (z-1)^{-1} \cdot 1 = \frac{1}{z-1},$$
$$\hat{H}(z) = \left[\begin{array}{cc|c} 1 & 2 & 1 \\ 0 & 3 & 0 \\ \hline 1 & 6 & 0 \end{array}\right] = \begin{bmatrix} 1 & 6 \end{bmatrix} \begin{bmatrix} z-1 & -2 \\ 0 & z-3 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{z-3}{z^2-4z+3} = \frac{1}{z-1}.$$

We can detect when two optimization algorithms are equivalent, even when one has additional (redundant) state variables, by computing their minimal realizations. This strategy shows that algorithms 2.3 and 2.4 are equivalent.

Inverse of state-space realization. Consider a state-space system $\mathbf{H}$ with realization (3.6) and for which $m = p$ (input and output dimension are the same). Is it possible to find a state-space system $\mathbf{H}^{-1}$ that maps $\mathbf{y}$ back to $\mathbf{u}$? It turns out this is possible if and only if D is invertible. In this case, the transfer function of $\mathbf{H}^{-1}$ is $\hat{H}^{-1}(z)$, a matrix whose entries are rational functions of z. We write the state-space realization of the inverse system $\mathbf{H}^{-1}$ as

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

This explicit realization can be obtained by applying the matrix inversion lemma to (3.11).

We can extend this idea to partial inverses of linear systems. Suppose the input sequence $\mathbf{u}$ is partitioned as

$$\mathbf{u} := (u^0, u^1, \dots) = \left(\begin{bmatrix} u_1^0 \\ u_2^0 \end{bmatrix}, \begin{bmatrix} u_1^1 \\ u_2^1 \end{bmatrix}, \dots \right) = \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{bmatrix}, \quad \text{where } u_1^k \in \mathbb{R}^{m_1}, u_2^k \in \mathbb{R}^{m_2} \text{ for all } k \geq 0$$

and similarly for $\mathbf{y}$. The matrix D and transfer matrix $\hat{H}(z)$ can also be partitioned conformally as

(3.13)
$$D = \begin{bmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{bmatrix} \quad \text{and} \quad \hat{H}(z) = \begin{bmatrix} \hat{H}_{11}(z) & \hat{H}_{12}(z) \\ \hat{H}_{21}(z) & \hat{H}_{22}(z) \end{bmatrix}, \quad \text{where } D_{ij} \in \mathbb{R}^{p_i \times m_j} \text{ and similarly for } \hat{H}(z).$$

If D_{11} is invertible, we can partially invert $\mathbf{H}$ with respect to $\mathbf{u}_1$ and $\mathbf{y}_1$ to form a new system $\mathbf{H}'$ that maps $(\mathbf{y}_1, \mathbf{u}_2) \mapsto (\mathbf{u}_1, \mathbf{y}_2)$. The transfer function $\hat{H}'(z)$ of the new system $\mathbf{H}'$ satisfies

$$\hat{H}'(z) = \begin{bmatrix} \hat{H}_{11}^{-1}(z) & -\hat{H}_{11}^{-1}(z)\hat{H}_{12}(z) \\ \hat{H}_{21}(z)\hat{H}_{11}^{-1}(z) & \hat{H}_{22}(z) - \hat{H}_{21}(z)\hat{H}_{11}^{-1}(z)\hat{H}_{12}(z) \end{bmatrix}. \tag{3.14}$$

A detailed proof of (3.14) is presented in appendix A. Note that if D_{22} is invertible, we can perform a similar partial inverse with respect to the second component.

When an optimization algorithm is related to another by conjugation of one of the function oracles, their transfer functions are related by (possibly partial) inversion.

4. Algorithm equivalence. We are now ready to revisit the motivating examples and formally define algorithm equivalence.

4.1. Oracle equivalence.

In the first motivating example, the algorithms have the same number of states, and the state sequences are equivalent up to an invertible linear transformation. We call these algorithms *state-equivalent*.

In the second motivating example, the state sequence of algorithm 2.3 can be transformed into the state sequence of algorithm 2.4 with a linear transformation. However, unlike the first motivating example, the linear transformation is not invertible; indeed, algorithm 2.4 uses fewer state variables than algorithm 2.3. Instead, recall that the sequence of calls to the gradient oracle are identical for algorithms 2.3 and 2.3. Hence these algorithms are *oracle-equivalent*.

DEFINITION 4.1. *Two algorithms are oracle-equivalent on a set of optimization problems if, for any problem in the set and for any initialization for one algorithm, there exists an initialization for the other such that the two algorithms generate the same oracle sequence.*

Notice that if the oracle sequences (that is, the oracles and their arguments y^k) are the same, then the oracles produce the same inputs u^k for the linear systems of each algorithm. Hence, as shown in figure 3, oracle-equivalent algorithms have matching input $\mathbf{u}$ and output $\mathbf{y}$ sequences. The solid double-sided arrow indicates the sequences y^k and $\tilde{y}^k$ are identical, and the sequences u^k and $\tilde{u}^k$ are identical.

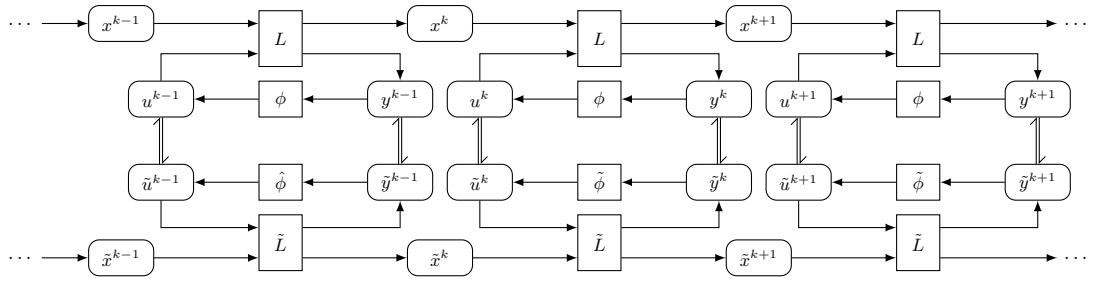

FIG. 3. *Unrolled block-diagram representation of oracle equivalence.*

Further, 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 $f(x)$ with $x \in R^n$, with solution $x^\star$ and corresponding objective value $f(x^\star)$. Further suppose f is convex and differentiable with oracle ∇f. If two algorithms are oracle-equivalent, the sequence of gradients $\|\nabla f(x)\|$, distance to the solution $\|x - x^\star\|$, and objective function values $\|f(x) - f(x^\star)\|$ evolve identically, so they have the same worst-case convergence, etc. 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.

4.2. Shift equivalence.

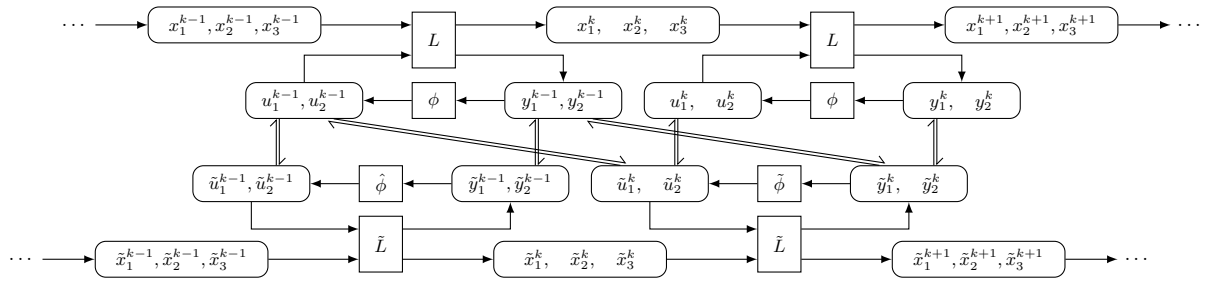

FIG. 4. *Unrolled block-diagram representation of shift equivalence.*

Now consider algorithms 2.5 and 2.6 from the third motivating example. They are not oracle-equivalent. However, their input and output sequences become identical after shifting algorithm 2.5 one step backward: these algorithms are *shift-equivalent*.

DEFINITION 4.2. *Two algorithms are shift-equivalent on a set of problems if, for any problem in the set and for any initialization for one algorithm, there exists an initialization for the other such that the oracle sequences match up to a prefix.*

Shift equivalence can also be interpreted as oracle equivalence up to a shift. We depict shift equivalence graphically in figure 4. Conversely, oracle equivalence can be regarded as a special case of shift equivalence, where the oracle sequences match without any shift.

4.3. Discussion.

One algorithm, many interpretations. Is it useful to have many different forms of an algorithm, if all the forms are (oracle- or shift-)equivalent? Yes: different rewritings of one algorithm often yield different ("physical") intuition. For example, algorithm 1.1 uses the current loss function for extrapolation [33]; while algorithm 1.2 seems to extrapolate from the previous loss function [7]. 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 [10,20]. In each time step k, algorithm 1.3 needs to store x_2^k, x_2^{k+1} and $F^k(\cdot)$, but algorithm 1.4 only needs to store x_1^k and x_1^{k+1} in memory.

Limitations. Do these formal notions of equivalence capture everything an optimization expert might mean by "equivalent algorithms"? No: an example is shown in algorithm 4.1. Algorithms 4.1 and 2.4 are related by a nonlinear state transformation, $x^k = \exp(\xi^k)$. However, none of the equivalences we have discussed capture this example. The difficulty is that algorithm 4.1 is a nonlinear algorithm, while all of our machinery for detecting algorithm equivalence requires linearity. While notions of nonlinear equivalence are certainly interesting, in this paper we will define only those types of equivalence that our framework can detect.

Algorithm 4.1

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

5. A characterization of oracle equivalence. In this section, we will discuss how to characterize oracle equivalence via transfer functions. Recall that oracle equivalence, introduced in section 4, characterizes an algorithm by its oracle sequence. This sequence is uniquely determined by the initialization of the algorithm (which we ignore) and the input-output map of the linear system representing the algorithm. While the state-space realization of two equivalent algorithms may differ, from subsection 3.3, recall that the transfer function of a linear system uniquely characterizes the system as an input-output map. Fortunately, using (3.11), we can directly calculate the transfer function from the state-space realization of an algorithm; and we can use equality of transfer functions to check if two algorithms are equivalent. This machinery allows us to avoid the issue of initialization (or of the optimization problem!) entirely, as we can check algorithm equivalence without ever producing a sequence of iterates.

More formally, consider two oracle-equivalent algorithms with the same number of oracle calls in each iteration. From subsection 4.1, we know that for every optimization problem, and for every initialization of the first algorithm, there exists an initialization of the second algorithm so that the oracle sequence of the two algorithms is the same. Concretely, by picking the initialization of the second algorithm appropriately, we can ensure that the first output of the linear systems match. Hence (since the oracles are the same), the first input of the linear systems match, and so the second output of the linear systems match, etc. By induction, for each possible sequence of input $\mathbf{u}$, they produce identical sequences of output $\mathbf{y}$. Then from subsection 3.3, the algorithms must have identical impulse responses and consequently identical transfer functions. In light of the previous discussion, we have proved the following proposition, since each step in the reasoning above is necessary and sufficient.

PROPOSITION 5.1. *Algorithms with the same number of oracle calls in each iteration are oracle-equivalent if and only if they have identical transfer functions.*

Importantly, oracle-equivalent algorithms have the same transfer function, even if they have a different number of state variables. But any realization of the algorithm must have at least as many state variables as the minimal realization of the linear system.

Oracle-equivalent algorithms have identical oracle sequences and hence converge to the same fixed point (if they converge). Suppose algorithm $\mathcal{A}_1 : \mathcal{X} \to \mathcal{X}$ with (nonlinear) oracle $\phi : \mathcal{X} \to \mathcal{X}$ and state-space realization (A_1, B_1, C_1, D_1), converges to a fixed point $(y^\star, u^\star, x^\star)$ that satisfies

$$\begin{aligned} x^\star &= A_1 x^\star + B_1 u^\star \\ y^\star &= C_1 x^\star + D_1 u^\star \\ u^\star &= \phi(y^\star). \end{aligned} \tag{5.1}$$

If algorithm $\mathcal{A}_2$ is oracle-equivalent to $\mathcal{A}_1$, $\mathcal{A}_2$ converges to a fixed point $(y^\star, u^\star, \tilde{x}^\star)$ that has the same output and input as the fixed point of $\mathcal{A}_1$; however, the state $\tilde{x}^\star$ may not be the same, or even have the same dimension.

Further, if there is an invertible linear map Q between the states of $\mathcal{A}_1$ and $\mathcal{A}_2$ and $(y^\star, u^\star, x^\star)$ is a fixed point of $\mathcal{A}_1$, then $(y^\star, u^\star, Qx^\star)$ is a fixed point of $\mathcal{A}_2$. We can use this fact to derive a relation between the state-space realizations of the two algorithms: the fixed point equation for $\mathcal{A}_2$ can be written as

$$\begin{aligned} Qx^\star &= QA_1Q^{-1}Qx^\star + QB_1u^\star \\ y^\star &= C_1Q^{-1}Qx^\star + D_1u^\star \\ u^\star &= \phi(y^\star), \end{aligned} \tag{5.2}$$

which shows that the state-space realization of $\mathcal{A}_2$ is

$$\begin{bmatrix} QA_1Q^{-1} & QB_1 \\ C_1Q^{-1} & D_1 \end{bmatrix}, \tag{5.3}$$

which can be obtained by (3.12).

5.1. Motivating examples: proof of equivalence. Now, we will revisit the first and second motivating examples and apply proposition 5.1 to show equivalence.

algorithm 2.1 and algorithm 2.2. The state-space realization and transfer function of algorithm 2.1 are shown as

$$\hat{H}_1(z) = \left[\begin{array}{cc|c} 2 & -1 & -\frac{1}{10} \\ 1 & 0 & 0 \\ \hline 2 & -1 & 0 \end{array}\right] = \begin{bmatrix} 2 & -1 \end{bmatrix} \left(zI - \begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix} \right)^{-1} \begin{bmatrix} -\frac{1}{10} \\ 0 \end{bmatrix} = \frac{-2z+1}{10(z-1)^2}.$$

The state-space realization and the transfer function of algorithm 2.2 are

$$\hat{H}_2(z) = \left[\begin{array}{cc|c} 1 & -1 & -\frac{1}{5} \\ 0 & 1 & \frac{1}{10} \\ \hline 1 & 0 & 0 \end{array}\right] = \begin{bmatrix} 1 & 0 \end{bmatrix} \left(zI - \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} -\frac{1}{5} \\ \frac{1}{10} \end{bmatrix} = \frac{-2z+1}{10(z-1)^2}.$$

Hence we see algorithms 2.1 and 2.2 have the same transfer function, so by proposition 5.1 they are oracle-equivalent. In fact, since the algorithms have the same number of state variables, there exists an invertible linear transformation

$$Q = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}$$

to convert the state-space realization of algorithm 2.1 to the state-space realization of algorithm 2.2 following (3.12).

algorithm 2.3 and algorithm 2.4. The state-space realization and transfer function of algorithm 2.3 are

$$\hat{H}_3(z) = \left[\begin{array}{cc|c} 3 & -2 & \frac{1}{5} \\ 1 & 0 & 0 \\ \hline -1 & 2 & 0 \end{array}\right] = \begin{bmatrix} -1 & 2 \end{bmatrix} \left(zI - \begin{bmatrix} 3 & -2 \\ 1 & 0 \end{bmatrix} \right)^{-1} \begin{bmatrix} \frac{1}{5} \\ 0 \end{bmatrix} = -\frac{1}{5(z-1)}.$$

The state-space realization and transfer function of algorithm 2.4 are

$$\hat{H}_4(z) = \left[\begin{array}{c|c} 1 & -\frac{1}{5} \\ \hline 1 & 0 \end{array}\right] = \begin{bmatrix} 1 \end{bmatrix} \left(zI - \begin{bmatrix} 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} -\frac{1}{5} \end{bmatrix} = -\frac{1}{5(z-1)}.$$

Algorithms 2.3 and 2.4 have the same transfer function, so by proposition 5.1 they are oracle-equivalent. On the other hand, they have different numbers of states. Consider the invertible linear transformation

$$Q = \begin{bmatrix} -1 & 2 \\ 0 & 1 \end{bmatrix}.$$

Applying Q to the state-space realization of algorithm 2.3 leads to

$$\left[\begin{array}{cc:c} 1 & 0 & -\frac{1}{5} \\ \hdashline -1 & 2 & 0 \\ 1 & 0 & 0 \end{array}\right],$$

where we have used dashed lines to demarcate the blocks in the state-space realization. This has the same minimal realization as algorithm 2.4 according to subsection 3.3 on minimal realizations.

$$\left[\begin{array}{c:c} 1 & -\frac{1}{5} \\ \hdashline 1 & 0 \end{array}\right].$$

Note that the state-space realization of algorithm 2.4 is a minimal realization. This shows the reason why algorithms 2.3 and 2.4 are equivalent even if they have different numbers of states.

Now we show how the sausage was made. Algorithm 2.3 was designed by starting with the more complex *Triple momentum algorithm* algorithm 5.1 [18, 32] and choosing parameters of the algorithm so its transfer function matched algorithm 2.4.

Algorithm 5.1 Triple momentum algorithm

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

The state-space realization and transfer function of algorithm 5.1 are

$$\hat{H}_7(z) = \left[\begin{array}{cc|c} 1+\beta & -\beta & -\alpha \\ 1 & 0 & 0 \\ \hline 1+\eta & -\eta & 0 \end{array}\right] = -\frac{\alpha((\eta+1)z - \eta)}{(z-1)(z-\beta)}. \tag{5.4}$$

We now demand that (5.4)(right) must equal the transfer function of algorithm 2.4 for all values of z, resulting in the equations

$$\begin{aligned} 5\alpha(\eta+1) &= 1 \\ 5\alpha\eta &= \beta. \end{aligned} \tag{5.5}$$

We solve for the parameters α, η and β to find a solution $\alpha = -\frac{1}{5}$, $\beta = 2$ and $\eta = -2$ to (5.5) that corresponds to algorithm 2.3. Other solutions exist: for example, $\alpha = 1$, $\beta = -4$ and $\eta = -\frac{4}{5}$ solves (5.5) and yields another (different!) algorithm equivalent to algorithm 2.4.

6. A characterization of shift equivalence. We can also characterize shift equivalence using transfer functions. Suppose an algorithm uses more than one oracle, and the call to the second oracle depends on the value of the first. Take algorithm 2.5 as example: at iteration k, the first update equation calls the oracle prox_f to compute $x_1^{k+1} = \text{prox}_f(x_3^k)$, and the second update equation calls the oracle prox_g to compute $x_2^{k+1} = \text{prox}_g(2x_1^{k+1} - x_3^k)$. This second update relies on the value of x_1^{k+1}. Imagine now that we reorder the update equations by some permutation. Generally this change produces an entirely different algorithm. But if the permutation is a *cyclic* permutation, the order of the oracle calls is preserved. In the example of algorithm 2.5, we could start with the update equation $x_2^{k+1} = \text{prox}_g(2x_1^{k+1} - x_3^k)$ and produce exactly the same sequence of oracle calls (after the first) by initializing x_1^{k+1} and x_3^k appropriately. This new algorithm is shift-equivalent to algorithm 2.5 by definition 4.2.

Algorithm 2.5 has three update equations, and so there are two other algorithms that may be produced by cyclic permutations of algorithm 2.5, shown below as algorithms 6.1 and 6.2.

Algorithm 6.1

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

Algorithm 6.2

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

Both are shift-equivalent to algorithm 2.5, but algorithm 6.2 is also oracle-equivalent to algorithm 2.5. (We will revisit and formally prove this result later.) It is easy to see why: the oracles prox_f and prox_g are called in the same order in algorithms 2.5 and 6.2, but in the opposite order in algorithm 6.1.

We introduce notation to generalize this idea to more complex algorithms. Consider an algorithm $\mathcal{A}$ that consists of m update equations and makes n sequential oracle calls in each iteration. We insist that no update equation may contain more than one oracle call, so $m \geq n$. At iteration k, the algorithm generates states $x_1^k, \ldots, x_m^k$, outputs $y_1^k, \ldots, y_n^k$, and inputs $u_1^k, \ldots, u_n^k$, respectively. Consider any permutation $\tilde{\pi}$ of the sequence $(m) = (1, \ldots, m)$. We call algorithm $\mathcal{B} = P_{\tilde{\pi}}\mathcal{A}$ a *permutation* of algorithm $\mathcal{A}$ if $\mathcal{B}$ performs the update equations of $\mathcal{A}$ in the order $\tilde{\pi}$ at each iteration. The algorithms $\mathcal{A}$ and $\mathcal{B}$ are shift-equivalent if and only if $\tilde{\pi}$ is a cyclic permutation of (m).

PROPOSITION 6.1. *An algorithm and any of its cyclic permutations are shift-equivalent.*

Proof. We provide a proof sketch here, and defer a detailed proof to appendix B. Let us name the oracle calls of the original algorithm $\mathcal{A}$ so that the oracles are called in order (n). Suppose $\mathcal{B} = P_{\tilde{\pi}}\mathcal{A}$ where $\tilde{\pi}$ is a cyclic permutation of (m). The permutation of update equations may reorder the oracle calls within one iteration, so that the oracle calls in algorithm $\mathcal{B}$ follow a cyclic permutation π of (n) (possibly, the identity). Hence $\mathcal{A}$ and $\mathcal{B}$ are shift-equivalent. (If the permutation is the identity, then the algorithms are also oracle-equivalent.) ∎

6.1. Reordering oracle calls. Most optimization algorithms proceed by sequential updates, each of which depends on the previous update. However, for completeness, we consider a more general class of equivalences that arises for algorithms whose oracle updates have a more complex dependency structure. We may express the order of oracle calls at each iteration using a directed graph, where the graph has edge from oracle i to oracle j if oracle call j depends on the result of oracle call i (within the same iteration). In other words, within the iteration we must call oracle i before oracle j. We call this directed graph the *oracle dependence graph* (ODG) of the algorithm.

An example is provided below as algorithm 6.3. Note that we are not aware of any practical algorithm for optimization with this ODG. It is constructed only for illustration.

Algorithm 6.3

for $k = 0, 1, 2, \ldots$ **do**
$x_1^{k+1} = x_4^k - t\nabla f(x_4^k)$
$x_2^{k+1} = x_1^{k+1} - t\nabla g(x_1^{k+1})$
$x_3^{k+1} = x_1^{k+1} - t\nabla h(x_1^{k+1})$
$x_4^{k+1} = \text{prox}_{tf}(x_2^{k+1} + x_3^{k+1})$
end for

Algorithm 6.4

for $k = 0, 1, 2, \ldots$ **do**
$x_1^{k+1} = x_4^k - t\nabla f(x_4^k)$
$x_3^{k+1} = x_1^{k+1} - t\nabla h(x_1^{k+1})$
$x_2^{k+1} = x_1^{k+1} - t\nabla g(x_1^{k+1})$
$x_4^{k+1} = \text{prox}_{tf}(x_2^{k+1} + x_3^{k+1})$
end for

Figure 5 expresses the dependency of oracle calls within each iteration of algorithm 6.3. At each iteration, oracle calls 2 (∇g) and 3 (∇h) depends on the result of oracle call 1 (∇f); oracle call 4 (prox_{tf}) depends on the results of oracle calls 1, 2, and 3.

An algorithm is always written as a sequence of update equations. But some algorithms might have a directed graph that may be written as a sequence (with all edges pointing forward) in more than one way. These algorithms can be implemented as a sequence of oracle calls in more than one way. For illustration,

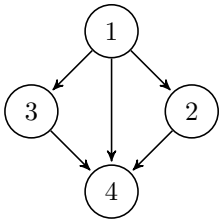

FIG. 5. *Directed graph representing dependency of oracle calls in algorithm 6.3.*

consider algorithms 6.3 and 6.4. At each iteration, the oracle calls of algorithms 6.3 and 6.4 are identical: that is, calls to oracles ∇f, ∇g, ∇h, and prox_{tf} are identical. The only difference is that the oracle calls ∇g and ∇h are swapped in the oracle sequence at each iteration. Notice that the state-space realizations of these algorithms *still* have the same transfer function (after swapping the second and third columns and rows). This is consistent with the fact that algorithms 6.3 and 6.4 share the same directed graph of oracle calls as figure 5.

We know of no practical optimization algorithm like this. However, were one to be discovered, we would suggest an expanded definition of oracle equivalence: two algorithms are oracle-equivalent if there exists a way of writing each algorithm as a sequence of updates so that both algorithms have the same sequence of oracle calls. We can still identify algorithms that are oracle-equivalent in this expanded sense using the transfer function.

The oracle calls in an algorithm at each iteration are always written in sequential form. This sequential form is lost in the state-space realization of the algorithm. However, the order (dependency) of oracle calls is encoded in the D matrix of the state-space realization. In this sense, the D matrix is closely related to the adjacency matrix of the directed graph. We have $D_{ij} \neq 0$ if and only if oracle call i depends on the results of oracle call j at each iteration. For example, the D matrix in the state-space realization of algorithm 6.3 is provided below.

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ -t & 0 & 0 & 0 \\ -t & 0 & 0 & 0 \\ -2t & -t & -t & 0 \end{bmatrix}$$

In light of this discussion, we can strengthen proposition 6.1 to proposition 6.2.

PROPOSITION 6.2. *An algorithm and any of its cyclic permutations are shift-equivalent; further, if they share the same D matrix in their state-space realizations, they are also oracle-equivalent.*

If an algorithm contains m update equations and n oracle calls at each iteration ($m \geq n$), there are m possible cyclic permutations on the update equations. According to the D matrix in the state-space realization, we can group the m cyclic permutations into n distinct equivalent classes. Algorithms within each equivalence class are oracle-equivalent and shift-equivalent, while algorithms in different equivalent classes are only shift-equivalent. The n distinct equivalence classes correspond to the n cyclic permutations of the original order of oracle calls (n).

6.2. Characterization of cyclic permutation.

In the remainder of this paper, let us restrict our attention to algorithms for which a (cyclic) permutation of the algorithm changes the update order of oracle calls within one iteration, or in other words, changes the D matrix in the state-space realization. In this way, we call algorithm $\mathcal{B} = P_\pi \mathcal{A}$ a permutation of algorithm $\mathcal{A}$ if $\mathcal{B}$ performs the update equations of $\mathcal{A}$ in a different order such that the update order of oracle calls of $\mathcal{B}$ is π at each iteration.

Suppose $\mathcal{A}$ has state-space realization (A, B, C, D), and $\mathcal{B} = P_\pi \mathcal{A}$ where $\pi = (j+1, \ldots, n, 1, \ldots, j)$ for $1 < j < n$ is a cyclic permutation of (n). We will show how to recognize this relationship between the algorithms by considering their transfer functions. Partition the oracle calls into two parts, $(1, \ldots, j)$ and $(j+1, \ldots, n)$, and partition the input and output sequences in the same way: $\bar{\mathbf{u}}_1$, $\bar{\mathbf{u}}_2$ for inputs and $\bar{\mathbf{y}}_1$, $\bar{\mathbf{y}}_2$ for outputs. The state-space realization $L_\mathcal{A}$ and transfer function $\hat{H}_\mathcal{A}(z)$ can also be partitioned accordingly

as

$$L_{\mathcal{A}} = \left[\begin{array}{c|cc} A & B_1 & B_2 \\ \hline C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \end{array}\right], \tag{6.1}$$

$$\hat{H}_{\mathcal{A}}(z) = \begin{bmatrix} C_1(zI-A)^{-1}B_1 + D_{11} & C_1(zI-A)^{-1}B_2 + D_{12} \\ C_2(zI-A)^{-1}B_1 + D_{21} & C_2(zI-A)^{-1}B_2 + D_{22} \end{bmatrix} = \begin{bmatrix} \hat{H}_{11}(z) & \hat{H}_{12}(z) \\ \hat{H}_{21}(z) & \hat{H}_{22}(z) \end{bmatrix}.$$

The state-space realization corresponds to the state update equations

$$\begin{aligned} x^{k+1} &= Ax^k + B_1\bar{u}_1^k + B_2\bar{u}_2^k \\ \bar{y}_1^k &= C_1x^k + D_{11}\bar{u}_1^k + D_{12}\bar{u}_2^k \\ \bar{y}_2^k &= C_2x^k + D_{21}\bar{u}_1^k + D_{22}\bar{u}_2^k. \end{aligned} \tag{6.2}$$

Now we can say how the transfer function of an algorithm is related to that of its cyclic permutation.

PROPOSITION 6.3. *Assume $D_{12} = 0$. Then $\mathcal{B} = P_\pi\mathcal{A}$ if and only if the transfer function of $\mathcal{B}$ satisfies*

$$\hat{H}_{\mathcal{B}}(z) = \begin{bmatrix} \hat{H}_{11}(z) & z\hat{H}_{12}(z) \\ \hat{H}_{21}(z)/z & \hat{H}_{22}(z) \end{bmatrix}. \tag{6.3}$$

Proof. Sufficiency. We will derive the state-space realization of $\mathcal{B}$:

$$\left[\begin{array}{cc|cc} A & B_1 & 0 & B_2 \\ 0 & 0 & I & 0 \\ \hline C_1A & C_1B_1 & D_{11} & C_1B_2 \\ C_2 & D_{21} & 0 & D_{22} \end{array}\right]. \tag{6.4}$$

To verify this realization is correct, we can write the system equations of this state-space realization as

$$\begin{aligned} x^{k+1} &= Ax^k + B_1\bar{u}_1^k + B_2\bar{u}_2^k \\ \bar{u}_1^{k+1} &= \bar{u}_1^{k+1} \\ \bar{y}_1^{k+1} &= C_1Ax^k + C_1B_1\bar{u}_1^k + D_{11}\bar{u}_1^{k+1} + C_1B_2\bar{u}_2^k \\ \bar{y}_2^k &= C_2x^k + D_{21}\bar{u}_1^k + D_{12}\bar{u}_2^k. \end{aligned} \tag{6.5}$$

Note that equations (6.5) are the results of equations (6.2) after applying permutation π. As we perform cyclic permutation π, within each iteration, the update order of the oracles is shifted as $(j+1,\ldots,n,1,\ldots,j)$, indicating oracles $(j+1,\ldots,n)$ are updated before $(1,\ldots,j)$. Further, the input and output sequences within one iteration at time step k become $(\bar{u}_2^k, \bar{u}_1^{k+1})$ and $(\bar{y}_2^k, \bar{y}_1^{k+1})$. From the state-space realization, we may compute the transfer function as

$$\hat{H}_{\mathcal{B}}(z) = \begin{bmatrix} C_1(zI-A)^{-1}B_1 + D_{11} & zC_1(zI-A)^{-1}B_2 \\ C_2(zI-A)^{-1}B_1/z + D_{21}/z & C_2(zI-A)^{-1}B_2 + D_{22} \end{bmatrix} = \begin{bmatrix} \hat{H}_{11}(z) & z\hat{H}_{12}(z) \\ \hat{H}_{21}(z)/z & \hat{H}_{22}(z) \end{bmatrix}. \tag{6.6}$$

To arrive at (6.6), we have used the fact that $D_{12} = 0$ by assumption, and

$$\left(zI - \begin{bmatrix} A & B_1 \\ 0 & 0 \end{bmatrix}\right)^{-1} = \begin{bmatrix} (zI-A)^{-1} & \frac{1}{z}(zI-A)^{-1}B_1 \\ 0 & \frac{1}{z}I \end{bmatrix}. \tag{6.7}$$

Necessity is provided by proposition 5.1. Equivalent algorithms must have identical transfer functions. Thus, if we find an algorithm and its transfer function is the same as (6.3), it must be equivalent to $\mathcal{B}$. ∎

We have assumed that $D_{12} = 0$ for algorithm $\mathcal{A}$. This assumption is quite weak. In fact, D_{12} must be 0 for any algorithm $\mathcal{A}$ that can be represented as a causal linear time-invariant system. Here, causal means that we can implement the algorithm by calling state update equations sequentially. To see this, suppose the state

update equations have been arranged in this order, and use (3.5) to write down the matrix representation of the infinite dimensional map $\mathbf{H}$ that maps input $\mathbf{u}$ to output $\mathbf{y}$ corresponding to $\mathcal{A}$ as (6.8):

$$\mathbf{H} = \begin{bmatrix} D_{11} & D_{12} & 0 & 0 & 0 & 0 & \cdots \\ D_{21} & D_{22} & 0 & 0 & 0 & 0 & \cdots \\ C_1B_1 & C_1B_2 & D_{11} & D_{12} & 0 & 0 & \cdots \\ C_2B_1 & C_2B_2 & D_{21} & D_{22} & 0 & 0 & \cdots \\ C_1AB_1 & C_1AB_2 & C_1B_1 & C_1B_2 & D_{11} & D_{12} & \cdots \\ C_2AB_1 & C_2AB_2 & C_2B_1 & C_2B_2 & D_{21} & D_{22} & \cdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \end{bmatrix}. \tag{6.8}$$

We can see that map $\mathbf{H}$ is (block) Toeplitz. Further, if algorithm $\mathcal{A}$ is causal, map $\mathbf{H}$ must be lower-triangular, and so D_{12} must be 0.

By causality, at each iteration the former oracle calls must be independent with the latter oracle calls while the latter calls can depend on the former calls. This indicates that there are no directed cycles in the directed graph representing oracle calls at each iteration for any causal algorithm. In other words, the graph is a directed acyclic graph (DAG). This is consistent with the fact that any causal algorithm has a lower-triangular D matrix (lower-triangular adjacency matrix of the directed graph).

Note that algorithms are not always written with state update equations ordered causally: for example, the state-space realization (6.4) has a non-zero D_{12} block. However, we may reorder these equations so that each equation depends only on previously-computed quantities to reveal that the iteration is causal; after this rearrangement, the new D_{12} block is 0. We discuss permutations further in appendix C.

The fixed points of an algorithm and its cyclic permutations are the same up to a permutation. Suppose algorithm $\mathcal{A} : \mathcal{X} \to \mathcal{X}$ with state-space realization of the form (6.1) converges to a fixed point $(\bar{y}_1^\star, \bar{y}_2^\star, \bar{u}_1^\star, \bar{u}_2^\star, x^\star)$. Partition the oracle calls into two (nonlinear) oracles ϕ_1 and ϕ_2. Formally, write the update equations as

$$\begin{aligned} x^\star &= Ax^\star + B_1\bar{u}_1^\star + B_2\bar{u}_2^\star \\ \bar{y}_1^\star &= C_1x^\star + D_{11}\bar{u}_1^\star + D_{12}\bar{u}_2^\star \\ \bar{y}_2^\star &= C_2x^\star + D_{21}\bar{u}_1^\star + D_{22}\bar{u}_2^\star \\ u_1^\star &= \phi_1(y_1^\star) \\ u_2^\star &= \phi_2(y_2^\star). \end{aligned} \tag{6.9}$$

Suppose the cyclic permutation π swaps the first and second set of oracle calls. Then the cyclic permutation $\mathcal{B} = P_\pi\mathcal{A}$ converges to fixed point $(\bar{y}_2^\star, \bar{y}_1^\star, \bar{u}_2^\star, \bar{u}_1^\star, x^\star)$. To verify this, since $D_{12} = 0$, we have

$$\begin{aligned} x^\star &= Ax^\star + B_1\bar{u}_1^\star + B_2\bar{u}_2^\star \\ \bar{u}_1^\star &= \bar{u}_1^\star \\ \bar{y}_1^\star &= C_1Ax^\star + C_1B_1\bar{u}_1^\star + D_{11}\bar{u}_1^\star + C_1B_2\bar{u}_2^\star = C_1x^\star + D_{11}\bar{u}_1^\star \\ \bar{y}_2^\star &= C_2x^\star + D_{21}\bar{u}_1^\star + D_{12}\bar{u}_2^\star \\ u_1^\star &= \phi_1(y_1^\star) \\ u_2^\star &= \phi_2(y_2^\star). \end{aligned} \tag{6.10}$$

6.3. Applications: proof of shift equivalence.

algorithm 2.5 *and algorithm* 2.6. Now, we can revisit algorithms 2.5 and 2.6 in the third motivating example and show that they are permutations of each other and they are shift-equivalent. The transfer function of algorithm 2.5 is

$$\hat{H}_5(z) = \left[\begin{array}{ccc|cc} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & -1 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 2 & 0 \end{array}\right] = \begin{bmatrix} -\frac{1}{z-1} & \frac{1}{z-1} \\ \frac{2z-1}{z-1} & -\frac{1}{z-1} \end{bmatrix}.$$

The transfer functions of algorithm 2.6 is

$$\hat{H}_6(z) = \left[\begin{array}{cc|cc} 1 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ \hline 1 & -1 & 0 & 1 \\ -1 & 2 & 0 & 0 \end{array}\right] = \left[\begin{array}{cc} -\frac{1}{z-1} & \frac{z}{z-1} \\ \frac{2z-1}{z(z-1)} & -\frac{1}{z-1} \end{array}\right].$$

From propositions 6.1 and 6.3, we know that they are (cyclic) permutation and they are shift-equivalent.

algorithm 6.1 and algorithm 6.2. Here we revisit algorithms 6.1 and 6.2 at the beginning of this chapter and show their relations with algorithm 2.5. The transfer function of algorithm 6.1 is

$$\hat{H}_8(z) = \left[\begin{array}{ccc|cc} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 1 & 0 & 1 \\ \hline -1 & 0 & 1 & 0 & 1 \\ 2 & 0 & -1 & 0 & 0 \end{array}\right] = \left[\begin{array}{cc} -\frac{1}{z-1} & \frac{z}{z-1} \\ \frac{2z-1}{z(z-1)} & -\frac{1}{z-1} \end{array}\right].$$

The transfer function of algorithm 6.2 is

$$\hat{H}_9(z) = \left[\begin{array}{ccc|cc} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & 1 & 0 & 0 \\ \hline -1 & 1 & 1 & 0 & 0 \\ 1 & -1 & -1 & 2 & 0 \end{array}\right] = \left[\begin{array}{cc} -\frac{1}{z-1} & \frac{1}{z-1} \\ \frac{2z-1}{z-1} & -\frac{1}{z-1} \end{array}\right].$$

From Propositions 6.1 and 6.3, we know that algorithms 2.5 and 6.1 are (cyclic) permutation and they are shift-equivalent. From Proposition 5.1, we know algorithms 2.5 and 6.2 are oracle-equivalent, thus they are also shift-equivalent.

Algorithm 6.5 Douglas-Rachford splitting

for $k = 0, 1, 2, \ldots$ **do**
$x_1^{k+1} = \mathrm{prox}_{tf}(x_3^k)$
$x_2^{k+1} = \mathrm{prox}_{tg}(2x_1^{k+1} - x_3^k)$
$x_3^{k+1} = x_3^k + x_2^{k+1} - x_1^{k+1}$
end for

Algorithm 6.6 ADMM

for $k = 0, 1, 2, \ldots$ **do**
$\xi_1^{k+1} = \mathrm{argmin}_\xi \{g(\xi) + \frac{\rho}{2} \left\| A\xi + B\xi_2^k - c + \xi_3^k \right\|^2\}$
$\xi_2^{k+1} = \mathrm{argmin}_\xi \{f(\xi) + \frac{\rho}{2} \left\| A\xi_1^{k+1} + B\xi - c + \xi_3^k \right\|^2\}$
$\xi_3^{k+1} = \xi_3^k + A\xi_1^{k+1} + B\xi_2^{k+1} - c$
end for

Douglas-Rachford splitting and ADMM. Consider a last example of algorithm permutation: Douglas-Rachford splitting (DR) (algorithm 6.5 [11,12]) and the alternating direction method of multipliers (ADMM) (algorithm 6.6 [29, §8]). Suppose that $A = I$, $B = -I$, and $c = 0$ in (3.1). Then both DR and ADMM solve problem (3.1) [4, 19, 34], and the update equations of ADMM can be simplified as algorithm 6.7. Further,

Algorithm 6.7 Simplified ADMM

for $k = 0, 1, 2, \ldots$ **do**
$\xi_1^{k+1} = \mathrm{prox}_{\frac{1}{\rho}g}(\xi_2^k - \xi_3^k)$
$\xi_2^{k+1} = \mathrm{prox}_{\frac{1}{\rho}f}(\xi_1^{k+1} + \xi_3^k)$
$\xi_3^{k+1} = \xi_3^k + \xi_1^{k+1} - \xi_2^{k+1}$
end for

we assume $\rho = 1/t$ in ADMM. We will compute the transfer function of both algorithms using prox_{tf} and prox_{tg} as the oracles. The transfer function of DR is

$$\hat{H}_{10}(z) = \left[\begin{array}{ccc|cc} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & -1 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 2 & 0 \end{array}\right] = \left[\begin{array}{cc} -\frac{1}{z-1} & \frac{1}{z-1} \\ \frac{2z-1}{z-1} & -\frac{1}{z-1} \end{array}\right] \tag{6.11}$$

and the transfer function of ADMM is

$$\hat{H}_{11}(z) = \left[\begin{array}{ccc|cc} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & 1 \\ \hline 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & -1 & 0 & 0 \end{array}\right] = \left[\begin{array}{cc} -\frac{1}{z-1} & \frac{z}{z-1} \\ \frac{2z-1}{z(z-1)} & -\frac{1}{z-1} \end{array}\right]. \tag{6.12}$$

From propositions 6.1 and 6.3, we know that Douglas-Rachford splitting and ADMM (with $\rho = 1/t$) are (cyclic) permutation and they are shift-equivalent. In fact, it is also possible to write the state-space realization for each algorithm using the gradient (or subgradient) of f and g as the oracle. The transfer functions depend on the choice of oracle, but in either case, we obtain the same results: the algorithms are (cyclic) permutation and they are shift-equivalent. We discuss the details further in appendix D.

7. Algorithm repetition. In previous sections, we define equivalence between algorithms with the same number of oracle calls in each iteration. This section considers how to identify relations between two algorithms when the number of oracles in each iteration differs. For example, we would like to detect when one algorithm consists of the another, simpler algorithm, repeated twice or more, possibly with changes to variables or shifts that obscure the relation.

Consider an algorithm $\mathcal{A}$. Given a problem and an initialization, the algorithm will generate state sequence $(x_{\mathcal{A}}^k)_{k\geq 0}$, input sequence $(u_{\mathcal{A}}^k)_{k\geq 0}$, and output sequence $(y_{\mathcal{A}}^k)_{k\geq 0}$, respectively. Specifically, the update at time step k can be written as $x_{\mathcal{A}}^{k+1} = \mathcal{A}(x_{\mathcal{A}}^k)$. Suppose we have another algorithm $\mathcal{B}$ such that $\mathcal{B} = \mathcal{A}^2$: repeating $\mathcal{A}$ twice gives the same result as $\mathcal{B}$. We call $\mathcal{B}$ a repetition of $\mathcal{A}$.

Just as in the previous sections, algorithm repetition can be characterized by the transfer function.

PROPOSITION 7.1. *Suppose $\mathcal{A}$ has state-space realization (A, B, C, D). Then $\mathcal{B} = \mathcal{A}^2$ if and only if its transfer function has the form*

$$\left[\begin{array}{cc} C(zI - A^2)^{-1}AB + D & C(zI - A^2)^{-1}B \\ CA(zI - A^2)^{-1}AB + CB & CA(zI - A^2)^{-1}B + D \end{array}\right]. \tag{7.1}$$

Proof. Sufficiency. The update equations of $\mathcal{B}$ can be written as

$$\begin{aligned} x_1^k &= Ax_{\mathcal{B}}^k + Bu_1^k \\ y_1^k &= Cx_{\mathcal{B}}^k + Du_1^k \\ x_{\mathcal{B}}^{k+1} &= Ax_1^k + Bu_2^k \\ y_2^k &= Cx_1^k + Du_2^k, \end{aligned} \tag{7.2}$$

where x_1^k is an intermediate state. After eliminating the intermediate state x_1^k, we arrive at a new system of update equations:

$$\begin{aligned} x_{\mathcal{B}}^{k+1} &= A^2x_{\mathcal{B}}^k + ABu_1^k + Bu_2^k \\ y_1^k &= Cx_{\mathcal{B}}^k + Du_1^k \\ y_2^k &= CAx_{\mathcal{B}}^k + CBu_1^k + Du_2^k. \end{aligned} \tag{7.3}$$

The corresponding state-space realization has transfer function

$$\left[\begin{array}{c|cc} A^2 & AB & B \\ \hline C & D & 0 \\ CA & CB & D \end{array}\right] = \left[\begin{array}{cc} C(zI - A^2)^{-1}AB + D & C(zI - A^2)^{-1}B \\ CA(zI - A^2)^{-1}AB + CB & CA(zI - A^2)^{-1}B + D \end{array}\right]. \tag{7.4}$$

Necessity is provided by proposition 5.1 since the transfer function uniquely characterizes an algorithm. ∎

Algorithm 7.1 Gradient method

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

Algorithm 7.2 Repetition of gradient method

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

One example of repetition consists the gradient method algorithm 7.1 and its repetition algorithm 7.2. Note that algorithm 2.4 is algorithm 7.1 with a specific parameter realization. The transfer functions of each algorithm are computed as $\hat{H}_{12}(z)$ and $\hat{H}_{13}(z)$ respectively:

$$\hat{H}_{12}(z) = \left[\begin{array}{c|c} 1 & -t \\ \hline 1 & 0 \end{array}\right] = -\frac{t}{z-1}, \qquad \hat{H}_{13}(z) = \left[\begin{array}{c|cc} 1 & -t & -t \\ \hline 1 & 0 & 0 \\ 1 & -t & 0 \end{array}\right] = \left[\begin{array}{cc} -\frac{t}{z-1} & -\frac{t}{z-1} \\ -\frac{tz}{z-1} & -\frac{t}{z-1} \end{array}\right].$$

Proposition 7.1 reveals how the transfer function changes when an algorithm is repeated twice. In fact, we can identify an algorithm that has been repeated arbitrarily many times. Suppose algorithm $\mathcal{C}$ is $\mathcal{A}$ repeated $n \geq 1$ times: $\mathcal{C} = \mathcal{A}^n$.

PROPOSITION 7.2. *Suppose $\mathcal{A}$ has state-space realization (A, B, C, D). Then $\mathcal{C} = \mathcal{A}^n$ for $n \geq 1$ if and only if $\mathcal{C}$ has a transfer function given by (7.6).*

Proof. Sufficiency. We can represent $\mathcal{C}$ with state-space realization

$$\left[\begin{array}{c:ccccc} A^n & A^{n-1}B & \dots & \dots & AB & B \\ \hdashline C & D & 0 & 0 & \dots & 0 \\ CA & CB & D & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ CA^{n-1} & CA^{n-2}B & \dots & \dots & CB & D \end{array}\right]. \tag{7.5}$$

Note that $(zI - A^n)^{-1}A^l = A^l(zI - A^n)^{-1}$ for any n and l. Let $\tilde{C} = C(zI - A^n)^{-1}$, and compute the transfer function of $\mathcal{C}$:

$$\left[\begin{array}{cccccc} \tilde{C}A^{n-1}B + D & \tilde{C}A^{n-2}B & \dots & \dots & \tilde{C}AB & \tilde{C}B \\ \tilde{C}A^nB + CB & \tilde{C}A^{n-1}B + D & \dots & \dots & \tilde{C}A^2B & \tilde{C}AB \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ \tilde{C}A^{2n-2}B + CA^{n-2}B & \tilde{C}A^{2n-3}B + CA^{n-3}B & \dots & \dots & \tilde{C}A^nB + CB & \tilde{C}A^{n-1}B + D \end{array}\right]. \tag{7.6}$$

Necessity is provided by proposition 5.1, just as in the proof of proposition 7.1. ∎

Remark. Proposition 7.1 is a special case of proposition 7.2 when $n = 2$. The dimension of transfer function of $\mathcal{C}$ is n times the dimension of transfer function of $\mathcal{A}$. Similarly, the dimension of input and output of $\mathcal{C}$ is n times the dimension of the input and output of $\mathcal{A}$. At time step k, we have $y_{\mathcal{C}}^k = (y_{\mathcal{A}}^{nk}, \dots, y_{\mathcal{A}}^{(n+1)k-1})$ and $u_{\mathcal{C}}^k = (u_{\mathcal{A}}^{nk}, \dots, u_{\mathcal{A}}^{(n+1)k-1})$.

Just as for oracle equivalence and cyclic permutations, the fixed points of an algorithm and its repetitions are related, as shown in proposition 7.3.

PROPOSITION 7.3. *If algorithm $\mathcal{A}$ converges to a fixed point $(y^\star, u^\star, x^\star)$, then its repetition $\mathcal{A}^n$ for $n \geq 1$ converges to fixed point $(y', u', x^\star)$, with $y' = y^\star \bigotimes \mathbb{1}^n$ and $u' = u^\star \bigotimes \mathbb{1}^n$. Here $\bigotimes$ is the Kronecker product and $\mathbb{1}^n$ is an n dimensional vector whose entries are all ones.*

Detailed proof is provided in appendix E. Since $\mathcal{A}^n$ repeats $\mathcal{A}$ n times, the input and output of the fixed point of $\mathcal{A}^n$ are obtained by repeating the input and output on the corresponding fixed point of $\mathcal{A}$ n times.

Repetition gives us many more ways to combine algorithms into complex and unwieldly (but convergent) new methods. We can repeat a sequence of iterations from different algorithms and regard them together as a new algorithm. Suppose we choose n algorithms $\mathcal{A}_1, \dots, \mathcal{A}_n$ with state-space realizations $(A_1, B_1, C_1, D_1), \dots, (A_n, B_n, C_n, D_n)$ and run one iteration of each as a single iteration of our new monster algorithm. For simplicity, suppose the state-space realization matrices A_i, B_i, C_i, D_i for each algorithm $\mathcal{A}_i$ have the same dimensions as all others $i = 1, \dots, n$. (Otherwise the result is harder to write down, but still straightforward to compute.) Then we can represent the resulting monster algorithm with transfer function

$$\left[\begin{array}{c|ccccc} \prod_{i=n}^1 A_i & \prod_{i=n}^2 A_iB_1 & \dots & \dots & A_nB_{n-1} & B_n \\ \hline C_1 & D_1 & 0 & 0 & \dots & 0 \\ C_2A_1 & C_2B_1 & D_2 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ C_n\prod_{i=n-1}^1 A_i & C_n\prod_{i=n-1}^2 A_iB_1 & \dots & \dots & C_nB_{n-1} & D_n \end{array}\right]. \tag{7.7}$$

Hence one easy way to develop new publishable optimization algorithms — until the present work — has been to combine existing algorithms into a new monster algorithm with similar convergence properties but new exciting interpretations. Using our software, it is easy for reviewers to detect such algorithm surgery by searching over all pairs (or trios, etc) of known algorithms. This combinatorial search is still not too expensive, since the list of known algorithms is still rather small, and the number of algorithms that makes up a monster algorithm is limited by the number of oracle calls at each iteration of the monster algorithm.

8. Algorithm conjugation. In this section, we introduce one last algorithm transformation, conjugation, which alters the oracle calls but results in algorithms that still bear a family resemblance.

Algorithm conjugation is a natural operation for convex optimization. For convex optimization, some oracles are closely related to others: for example, when $f^\star(y) = \sup_x\{x^T y - f(x)\}$ is the Fenchel conjugate of f [13, §3],

- $(\partial f)^{-1} = \partial f^\star$, and
- *Moreau's identity.* $I - \text{prox}_f = \text{prox}_{f^*}$

[28; 29, §2]. 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 [2, §10; 3]) by rewriting it in terms of the conjugate of the original oracle prox_g, using Moreau's identity, as algorithm 8.2 [22].

Algorithm 8.1 Proximal gradient method

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

Algorithm 8.2 Conjugate of proximal gradient method

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

The transfer function of the algorithm changes when we rewrite the algorithm to call a different oracle, such as calling $\text{prox}_{f^\star}$ instead of prox_f. Yet the sequence of states is preserved! Similarly, when we rewrite an algorithm to call $\partial f^\star$ instead of ∂f, the resulting algorithm is related to the original algorithm by swapping the input and output sequences. We say that algorithm $\mathcal{B} = \mathcal{C}_\kappa\mathcal{A}$ is a conjugate of algorithm $\mathcal{A}$ if algorithm $\mathcal{B}$ results from rewriting algorithm $\mathcal{A}$ to use the conjugates of the oracles in set $\kappa \subseteq [n]$, where $[n] = \{1, \ldots, n\}$ is the set of oracle indices for algorithm $\mathcal{A}$. Interestingly, conjugation preserves the state sequence but not the oracle sequence. We will also call two algorithms conjugates if they are oracle-equivalent to a conjugate pair. Our goal in this section is to describe how to identify conjugate algorithms.

For simplicity in the remainder of this section, we suppose that all oracles are (sub)gradients. To detect equivalence of algorithms involving prox using methods presented here, we may write the state-space realization of the algorithm in terms of (sub)gradients:

$$u = \text{prox}_f(y) \quad \iff \quad y \in u + \partial f(u).$$

In fact, our software uses this method to check algorithm conjugation.

Restricting to (sub)gradients, we see from the identity $(\partial f)^{-1} = \partial f^\star$ that algorithm conjugation swaps the input and output of an algorithm: the algorithm after conjugation takes the output of the original algorithm as input and produces the input of the original one as output. As shown in figure 6, the input sequence of the algorithm after conjugation is the original output sequence and the output sequence in the algorithm after conjugation is the original input sequence.

First, let's introduce a bit of standard notation. Suppose an algorithm $\mathcal{A}$ contains n oracle calls in each iteration. The cardinality of a subset $\kappa \subseteq [n]$ is $|\kappa|$ and the complement is $\bar{\kappa} = [n] \setminus \kappa$. For any matrix $M \in \mathbb{R}^{n\times n}$, $M[\kappa, \nu]$ is the sub-matrix of M whose rows and columns are indexed by κ and $\nu \subseteq [n]$, respectively. We write $M[\kappa, \kappa]$ as $M[\kappa]$ for simplicity. For $i \in [n]$, the conjugation operator $\mathcal{C}_i$ conjugates oracle i: it replaces the ith oracle by its inverse. The operator $\mathcal{C}_\kappa$ conjugates all oracles in the set $\kappa \subseteq [n]$ to produce the conjugate algorithm $\mathcal{C}_\kappa\mathcal{A}$.

PROPOSITION 8.1. *Suppose $\mathcal{A}$ has state-space realization (A, B, C, D) and transfer function $\hat{H}(z)$, and $D[\kappa]$ is invertible. Then $\mathcal{B} = \mathcal{C}_\kappa\mathcal{A}$ if and only if the transfer function $\hat{H}'(z)$ of $\mathcal{B}$ satisfies*

$$P\hat{H}'(z)P^T = \begin{bmatrix} \hat{H}[\kappa]^{-1}(z) & -\hat{H}[\kappa]^{-1}(z)\hat{H}[\kappa, \bar{\kappa}](z) \\ \hat{H}[\bar{\kappa}, \kappa](z)\hat{H}[\kappa]^{-1}(z) & \hat{H}[\bar{\kappa}](z) - \hat{H}[\bar{\kappa}, \kappa](z)\hat{H}[\kappa]^{-1}(z)\hat{H}[\kappa, \bar{\kappa}](z) \end{bmatrix}. \tag{8.1}$$

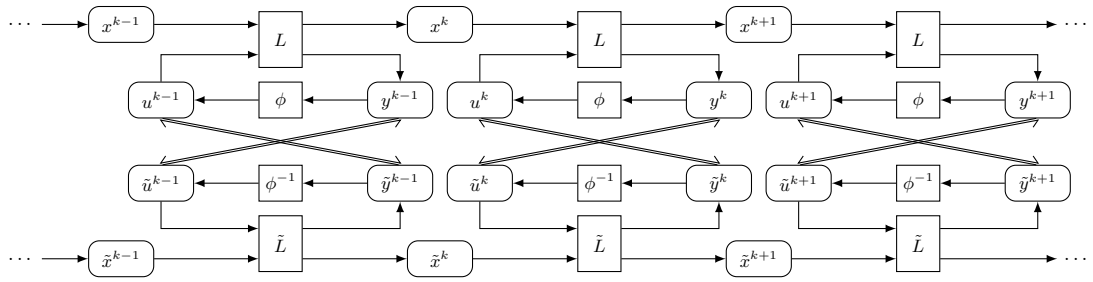

FIG. 6. *Unrolled block-diagram representation of algorithm conjugation. Here* $\tilde{\phi} = \phi^{-1}$.

Here P is a permutation matrix that swaps rows and columns so indices in κ come first:

$$P\hat{H}(z)P^T = \begin{bmatrix} \hat{H}[\kappa](z) & \hat{H}[\kappa,\bar{\kappa}](z) \\ \hat{H}[\bar{\kappa},\kappa](z) & \hat{H}[\bar{\kappa}](z) \end{bmatrix}. \tag{8.2}$$

Proof. Sufficiency. Without loss of generality, suppose the oracles $\kappa = \{1,\dots,|\kappa|\}$ appear first,

$$\hat{H}(z) = \begin{bmatrix} \hat{H}[\kappa](z) & \hat{H}[\kappa,\bar{\kappa}](z) \\ \hat{H}[\bar{\kappa},\kappa](z) & \hat{H}[\bar{\kappa}](z) \end{bmatrix}, \qquad D = \begin{bmatrix} D[\kappa] & D[\kappa,\bar{\kappa}] \\ D[\bar{\kappa},\kappa] & D[\bar{\kappa}] \end{bmatrix},$$

and consequently the permutation matrix P is the identity. We obtain the desired results from (3.14) by setting $D_{11} = D[\kappa]$, $\hat{H}_{11}(z) = \hat{H}[\kappa](z)$, $\hat{H}_{12}(z) = \hat{H}[\kappa,\bar{\kappa}](z)$, $\hat{H}_{21}(z) = \hat{H}[\bar{\kappa},\kappa](z)$, and $\hat{H}_{22}(z) = \hat{H}[\bar{\kappa}](z)$.

Necessity is provided by proposition 5.1 as the transfer function uniquely characterizes oracle-equivalent algorithms. ∎

From proposition 8.1, the transfer function $\hat{H}(z)$ of algorithm $\mathcal{A}$ is partially inverted when the algorithm is conjugated by $\mathcal{C}_\kappa$. The new transfer function $\hat{H}'(z)$ results from applying the Sweep operator with indices κ to $\hat{H}(z)$ [15, 31]. If we consider the input and output sequences for each oracle separately, for any oracle in κ, the input sequence corresponding to $\mathcal{C}_\kappa\mathcal{A}$ is the original output sequence in $\mathcal{A}$ and the output sequence corresponding to $\mathcal{C}_\kappa\mathcal{A}$ is the original input sequence in $\mathcal{A}$. The input and output sequences of oracles in $[n]\setminus\kappa$ remain unchanged in the new algorithm $\mathcal{C}_\kappa\mathcal{A}$.

Proposition 8.1 assumes that $D[\kappa]$ is invertible. In fact, $\mathcal{B} = \mathcal{C}_\kappa\mathcal{A}$ is a causal algorithm if and only if $D[\kappa]$ is invertible. We need not condition on causality in the proposition, since any algorithm that can be written down as a set of update equations is necessarily causal.

Now we consider two special cases: conjugating 1) a single oracle, or 2) all of the oracles.

COROLLARY 8.2. *Consider algorithm $\mathcal{A}$ with state-space realization (A,B,C,D) and transfer function $\hat{H}(z) \in \mathbb{R}^{n\times n}$.*

(a) Suppose $D_{kk} \neq 0$ for any $k \in [n]$. Then the new transfer function $\hat{H}'(z)$ of $\mathcal{C}_k\mathcal{A}$ can be expressed entrywise as

$$h'_{ij}(z) = \begin{cases} 1/h_{kk}(z) & i = k,\ j = k \\ -h_{kj}(z)/h_{kk}(z) & i = k,\ j \neq k \\ h_{ik}(z)/h_{kk}(z) & i \neq k,\ j = k \\ h_{ij}(z) - h_{ik}(z)h_{kj}(z)/h_{kk}(z) & i \neq k,\ j \neq k, \end{cases} \tag{8.3}$$

as $h_{ij}(z)$ and $h'_{ij}(z)$ $1 \leq i,j \leq n$ denote the entries of $\hat{H}(z)$ and $\hat{H}'(z)$ respectively.

(b) Suppose D is invertible. Then the transfer function $\hat{H}'(z)$ of $\mathcal{C}_{[n]}\mathcal{A}$ satisfies $\hat{H}'(z) = \hat{H}^{-1}(z)$.

Proximal gradient. Now we can revisit algorithms 8.1 and 8.2 and show that they are conjugate. The transfer functions of algorithms 8.1 and 8.2 are computed as $\hat{H}_{14}(z)$ and $\hat{H}_{15}(z)$ below. Note that the state-space realizations are written in terms of (sub)gradients. From corollary 8.2, they are conjugate with respect to the second oracle.

$$\hat{H}_{14}(z) = \begin{bmatrix} -\frac{t}{z-1} & -\frac{t}{z-1} \\ -\frac{tz}{z-1} & -\frac{tz}{z-1} \end{bmatrix}, \qquad \hat{H}_{15}(z) = \begin{bmatrix} 0 & \frac{1}{z} \\ -1 & -\frac{z-1}{tz} \end{bmatrix}$$

Algorithm 8.3 Chambolle-Pock method

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

DR and Chambolle-Pock. Another important example is the relation between DR (algorithm 6.5) and the primal-dual optimization method proposed by Chambolle and Pock (algorithm 8.3 [8; 24]). Note that algorithm 6.5 is parameterized by parameter t and algorithm 8.3 is parameterized by parameters τ and σ. The transfer functions of algorithms 6.5 and 8.3 are provided below as $\hat{H}_{10}(z)$ and $\hat{H}_{16}(z)$ respectively. By corollary 8.2, we know that they are conjugate with respect to the second oracle if $\tau = t$ and $\sigma = 1/t$. So DR and the Chambolle-Pock method (when the parameter value $\tau = t$ and $\sigma = 1/t$) are conjugate. We will say more about how to discover the correct parameter restriction in section 9.

$$\hat{H}_{10}(z) = \begin{bmatrix} -\frac{tz}{z-1} & -\frac{t}{z-1} \\ \frac{t(1-2z)}{z-1} & -\frac{tz}{z-1} \end{bmatrix}, \quad \hat{H}_{16}(z) = \begin{bmatrix} -\frac{\tau z(z-1)}{2\sigma\tau z - \sigma\tau + z^2 - 2z + 1} & \frac{\sigma\tau z}{2\sigma\tau z - \sigma\tau + z^2 - 2z + 1} \\ \frac{\sigma\tau z(1-2z)}{2\sigma\tau z - \sigma\tau + z^2 - 2z + 1} & -\frac{\sigma z(z-1)}{2\sigma\tau z - \sigma\tau + z^2 - 2z + 1} \end{bmatrix} \xrightarrow[\sigma=\frac{1}{t}]{\tau=t} \begin{bmatrix} \frac{t(1-z)}{z} & \frac{1}{z} \\ \frac{1-2z}{z} & \frac{1-z}{tz} \end{bmatrix}$$

The fixed points of an algorithm and its conjugate are related as stated in proposition 8.3.

PROPOSITION 8.3. *If an algorithm $\mathcal{A}$ converges to a fixed point $(y[\kappa]^\star, y[\bar{\kappa}]^\star, u[\kappa]^\star, u[\bar{\kappa}]^\star, x^\star)$, then its conjugate $\mathcal{B} = \mathcal{C}_\kappa \mathcal{A}$ converges to fixed point $(u[\kappa]^\star, y[\bar{\kappa}]^\star, y[\kappa]^\star, u[\bar{\kappa}]^\star, x^\star)$.*

For simplicity, detailed proof is provided in appendix F. Intuitively, as we invert the input-output map of $u[\kappa]$ and $y[\kappa]$, the corresponding parts in the fixed point are also inverted.

PROPOSITION 8.4. *Suppose algorithm $\mathcal{A}$ has state-space realization (A, B, C, D), where $D_{ii} \neq 0$ and $D_{jj} \neq 0$. Then $\mathcal{C}_i\mathcal{C}_j\mathcal{A} = \mathcal{C}_j\mathcal{C}_i\mathcal{A} = \mathcal{C}_{\{ij\}}\mathcal{A}$.*

Proof. By corollary 8.2, if $D_{ii} \neq 0$ and $D_{jj} \neq 0$, then $\mathcal{C}_i\mathcal{A}$ and $\mathcal{C}_j\mathcal{A}$ are causal. Note that entries above diagonal of D are all zero because $\mathcal{A}$ is causal. Thus, $\det(D[\{ij\}]) = D_{ii}D_{jj} \neq 0$ and $\mathcal{C}_{\{ij\}}\mathcal{A}$ is causal. The commutative property of the Sweep operator gives the result $\mathcal{C}_i\mathcal{C}_j\mathcal{A} = \mathcal{C}_j\mathcal{C}_i\mathcal{A} = \mathcal{C}_{\{ij\}}\mathcal{A}$ [15, 31]. ∎

Proposition 8.4 states that conjugation of different oracles commutes. This justifies our notation $\mathcal{C}_\kappa$ for set κ, as the order of the oracles in κ is irrelevant. Further, conjugation and cyclic permutation also commute; see proposition G.1 and proof in appendix G.

DR and ADMM. We showed in subsection 6.3 that the DR (algorithm 6.5) and ADMM (algorithm 6.6) are related by permutation with a certain choice of parameters. Here, we show that they are related by permutation and conjugation (in either order, as they commute), with a different choice of parameters: $A = I, B = I, c = 0, \rho = t$ for ADMM. The transfer function of this special parameterization of ADMM is shown as $\hat{H}_{17}(z)$. Relations between DR and ADMM can be illustrated as follows. Recall $\hat{H}_{10}(z)$ is the transfer function of DR. Here we can observe that different choices of parameters of algorithms can lead to different relations between algorithms.

$$\hat{H}_{17}(z) = \begin{bmatrix} -\frac{z}{t(z-1)} & \frac{2z-1}{tz(z-1)} \\ \frac{z}{t(z-1)} & -\frac{z}{t(z-1)} \end{bmatrix} \xrightarrow{\mathcal{C}_{12}} \begin{bmatrix} -\frac{tz}{z-1} & \frac{t(1-2z)}{z(z-1)} \\ -\frac{tz}{z-1} & -\frac{tz}{z-1} \end{bmatrix} \xrightarrow{P_{21}} \begin{bmatrix} -\frac{tz}{z-1} & -\frac{t}{z-1} \\ \frac{t(1-2z)}{z-1} & -\frac{tz}{z-1} \end{bmatrix} = \hat{H}_{10}(z)$$

The commutative property is important to identify relations between algorithms efficiently. For example, suppose we would like to identify the relations between algorithms 6.5 and 6.6, with transfer functions $\hat{H}_{10}(z)$ and $\hat{H}_{17}(z)$. We can first perform conjugation and next permutation on algorithm 6.5, and then test equivalence between the resulting algorithm and algorithm 6.6. We need not try permutation followed by conjugation; as these commute, both orders lead to the same transfer function.

We have already shown several relations between DR (algorithm 6.5), ADMM (algorithm 6.6), and the Chambolle-Pock method (algorithm 8.3) using conjugation and permutation. We represent these relations in figure 7. The figure relates 8 different algorithms: Starting from DR, since it contains 2 oracles, there are 2 possible different algorithms by permutation. From the state-space realization, we can conjugate both oracles, which yields 4 different algorithms by conjugation of different oracles. Therefore, in total there are $2 \times 4 = 8$ possible different algorithms, including both ADMM and Chambolle-Pock. In the figure, $\mathcal{C}_1$ and

$\mathcal{C}_2$ denote conjugation with respect to the first and second oracles respectively, P denotes permutation, and we can move between algorithms by applying the transformation on each edge, in either direction, as each transformation is an involution.

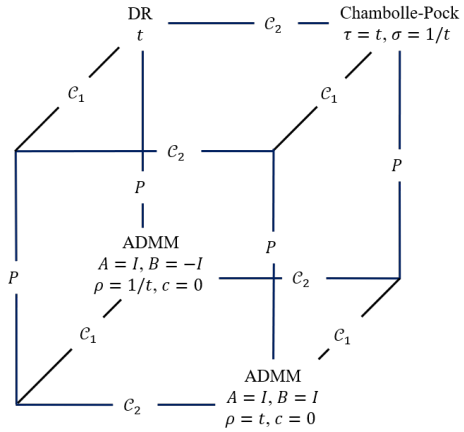

FIG. 7. *Connections between DR, ADMM, and Chambolle-Pock method.*

9. Linnaeus. We have presented a framework for detecting equivalence between iterative algorithms for continuous optimization. In this section, we introduce a software package called LINNAEUS that implements these ideas. This package can be used by researchers (or peer reviewers) who wish to understand the novelty of new algorithmic ideas and connections to existing algorithms. The input is an algorithm described in user-friendly syntax with variables, parameters, functions, oracles, and update equations. The system will automatically translate the input algorithm into a canonical form (the transfer function) and use the canonical form to identify whether the algorithm is equivalent to any reference algorithm, possibly after transformations such as permutation, conjugation, or repetition. Further, the software can also serve as a search engine, which will identify connections from the input algorithm to existing algorithms in the literature that appear in LINNAEUS's algorithm library.

9.1. Illustrative examples. We use LINNAEUS to identify the relations between algorithms presented previously in the paper. These examples demonstrate the power and simplicity of LINNAEUS. Code for these examples can be found at https://github.com/QCGroup/linnaeus.

algorithm 2.1 *and algorithm* 2.2. The following code identifies that algorithms 2.1 and 2.2 are oracle-equivalent. We input algorithms 2.1 and 2.2 with variables, oracles, and update equations, and parse them into state-space realizations. Then we check oracle equivalence using the function `is_equivalent`. The system returns `True`, consistent with our analytical results in sections 2 and 5.

```
# define Algorithm 2.1
algo1 = Algorithm("Algorithm 2.1")

# add oracle gradient of f to Algorithm 2.1
gradf = algo1.add_oracle("gradf")

# add variables x1, x2, and x3 to Algorithm 2.1
x1, x2, x3 = algo1.add_var("x1", "x2", "x3")

# add update equations
# x3 <- 2x1 - x2
algo1.add_update(x3, 2*x1 - x2)
# x2 <- x1
algo1.add_update(x2, x1)
# x1 <- x3 - 1/10*gradf(x3)
```

```
algo1.add_update(x1, x3 - 1/10*gradf(x3))

# parse Algorithm 2.1, translate it into canonical form
algo1.parse()
```

```
------------------------------------------------------------------
Parse Algorithm 2.1.
State-space realization:
```

$$\begin{bmatrix} x_1^+ \\ x_2^+ \\ x_3^+ \end{bmatrix} = \begin{bmatrix} 2 & -1 & 0 \\ 1 & 0 & 0 \\ 2 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} -0.1 \\ 0 \\ 0 \end{bmatrix} \left[\text{gradf}\,(y_0)\right]$$

$$\left[y_0\right] = \begin{bmatrix} 2 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \left[0\right] \left[\text{gradf}\,(y_0)\right]$$

```
------------------------------------------------------------------
```

```
algo2 = Algorithm("Algorithm 2.2")
xi1, xi2, xi3 = algo2.add_var("xi1", "xi2", "xi3")
gradf = algo2.add_oracle("gradf")

# xi3 <- xi1
algo2.add_update(xi3, xi1)
# xi1 <- xi1 - xi2 - 1/5*gradf(xi1)
algo2.add_update(xi1, xi1 - xi2 - 1/5*gradf(xi3))
# xi2 <- xi2 + 1/10*gradf(xi3)
algo2.add_update(xi2, xi2 + 1/10*gradf(xi3))

algo2.parse()
```

```
------------------------------------------------------------------
Parse Algorithm 2.2.
State-space realization:
```

$$\begin{bmatrix} \xi_1^+ \\ \xi_2^+ \\ \xi_3^+ \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \end{bmatrix} + \begin{bmatrix} -0.2 \\ 0.1 \\ 0 \end{bmatrix} \left[\text{gradf}\,(y_0)\right]$$

$$\left[y_0\right] = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \end{bmatrix} + \left[0\right] \left[\text{gradf}\,(y_0)\right]$$

```
------------------------------------------------------------------
```

```
# check oracle equivalence
lin.is_equivalent(algo1, algo2, verbose = True)
```

```
------------------------------------------------------------------
Algorithm 2.1 is equivalent to Algorithm 2.2.
------------------------------------------------------------------
True
```

algorithm 2.5 *and algorithm* 2.6. The second example identifies that algorithms 2.5 and 2.6 are shift-equivalent. We input and parse the algorithms into state-space realizations and then check shift equivalence (cyclic permutation) using the function `is_permutation`. The system returns `True`, consistent with results in sections 2 and 6.

```
algo5 = Algorithm("Algorithm 2.5")
x1, x2, x3 = algo5.add_var("x1", "x2", "x3")
proxf, proxg = algo5.add_oracle("proxf", "proxg")
```

```
# x1 <- proxf(x3)
algo5.add_update(x1, proxf(x3))
# x2 <- proxg(2x1 - x3)
algo5.add_update(x2, proxg(2*x1 - x3))
# x3 <- x3 + x2 - x1
algo5.add_update(x3, x3 + x2 - x1)

algo5.parse()
```

```
------------------------------------------------------------
Parse Algorithm 2.5.
State-space realization:
```

$$\begin{bmatrix} x_1^+ \\ x_2^+ \\ x_3^+ \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} \text{prox}_f(y_0) \\ \text{prox}_g(y_1) \end{bmatrix}$$

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} \text{prox}_f(y_0) \\ \text{prox}_g(y_1) \end{bmatrix}$$

```
------------------------------------------------------------
```

```
algo4 = Algorithm("Algorithm 2.6")
xi1, xi2 = algo4.add_var("xi1", "xi2")
proxf, proxg = algo4.add_oracle("proxf", "proxg")

# xi1 <- proxg(-xi1 + 2xi2) + xi1 - xi2
algo4.add_update(xi1, proxg(-xi1 + 2*xi2) + xi1 - xi2)
# xi2 <- proxf(xi1)
algo4.add_update(xi2, proxf(xi1))

algo4.parse()
```

```
------------------------------------------------------------
Parse Algorithm 2.6.
State-space realization:
```

$$\begin{bmatrix} \xi_1^+ \\ \xi_2^+ \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \text{prox}_g(y_0) \\ \text{prox}_f(y_1) \end{bmatrix}$$

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} -1 & 2 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \text{prox}_g(y_0) \\ \text{prox}_f(y_1) \end{bmatrix}$$

```
------------------------------------------------------------
```

```
# check cyclic permutation (shift equivalence)
lin.is_permutation(algo5, algo6, verbose = True)
```

```
------------------------------------------------------------
Algorithm 2.5 is a permutation of Algorithm 2.6.
------------------------------------------------------------
True
```

DR and ADMM. The third illustrative example shows that DR and ADMM are related by permutation and conjugation, as we saw in section 8. Further, LINNAEUS can even reveal the specific parameter choice required for the relation to hold. Just as in section 8, suppose both DR and ADMM solve problem (3.1) with $A = I$, $B = I$, and $c = 0$. We input and parse DR and ADMM. To detect the relations, we use function `test_conjugate_permutation` to check conjugation and permutation between DR and ADMM. The results are the same as section 8.

```
DR = Algorithm("Douglas-Rachford splitting")
x1, x2, x3 = DR.add_var("x1", "x2", "x3")
t = DR.add_parameter("t")

# x1 <- prox_tf(x3)
DR.add_update(x1, lin.prox(f, t)(x3))
# x2 <- prox_tg(2x1 - x3)
DR.add_update(x2, lin.prox(g, t)(2*x1 - x3))
# x3 <- x3 + x2 - x1
DR.add_update(x3, x3 + x2 - x1)

DR.parse()
```

```
---------------------------------------------------------------
Parse Douglas-Rachford splitting.
State-space realization:
```

$$\begin{bmatrix} x_1^+ \\ x_2^+ \\ x_3^+ \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} -t & 0 \\ -2t & -t \\ -t & -t \end{bmatrix} \begin{bmatrix} \frac{d}{dy_0} f(y_0) \\ \frac{d}{dy_1} g(y_1) \end{bmatrix}$$

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} -t & 0 \\ -2t & -t \end{bmatrix} \begin{bmatrix} \frac{d}{dy_0} f(y_0) \\ \frac{d}{dy_1} g(y_1) \end{bmatrix}$$

```
---------------------------------------------------------------
```

```
ADMM = Algorithm("ADMM")
f, g = ADMM.add_function("f", "g")
rho = ADMM.add_parameter("rho")
xi1, xi2, xi3 = ADMM.add_var("xi1", "xi2", "xi3")

# xi1 <- argmin(x1, g^*(xi1) + 1/2*rho*||xi1 + xi2 + xi3||^2)
ADMM.add_update(xi1, lin.argmin(xi1, g(xi1) + 1/2*rho*lin.norm_square(xi1 +␣
↪xi2 + xi3)))
# xi2 <- argmin(x2, f^*(xi2) + 1/2*rho*||xi1 + xi2 + xi3||^2)
ADMM.add_update(xi2, lin.argmin(xi2, f(xi2) + 1/2*rho*lin.norm_square(xi1 +␣
↪xi2 + xi3)))
# xi3 <- xi3 + xi1 + xi2
ADMM.add_update(xi3, xi3 + xi1 + xi2)

ADMM.parse()
```

```
---------------------------------------------------------------
Parse ADMM.
State-space realization:
```

$$\begin{bmatrix} \xi_1^+ \\ \xi_2^+ \\ \xi_3^+ \end{bmatrix} = \begin{bmatrix} 0 & -1 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \end{bmatrix} + \begin{bmatrix} -\frac{1}{\rho} & 0 \\ \frac{1}{\rho} & -\frac{1}{\rho} \\ 0 & -\frac{1}{\rho} \end{bmatrix} \begin{bmatrix} \frac{d}{dy_0} g(y_0) \\ \frac{d}{dy_1} f(y_1) \end{bmatrix}$$

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \end{bmatrix} + \begin{bmatrix} -\frac{1}{\rho} & 0 \\ \frac{1}{\rho} & -\frac{1}{\rho} \end{bmatrix} \begin{bmatrix} \frac{d}{dy_0} g(y_0) \\ \frac{d}{dy_1} f(y_1) \end{bmatrix}$$

```
---------------------------------------------------------------
```

```
# check permutation and conjugation
# between DR and ADMM
lin.test_conjugate_permutation(DR, ADMM)
```

```
----------------------------------------------------------------
================================================================
Parameters of Douglas-Rachford splitting:
 t
Parameters of ADMM:
 ρ
Douglas-Rachford splitting is a conjugate permutation of ADMM, if the parameters
satisfy:
 ρ = t
================================================================
----------------------------------------------------------------
```

9.2. Implementation. In this subsection, we briefly describe the implementation of LINNAEUS. All expressions in LINNAEUS are defined symbolically, using the python package for symbolic mathematics *sympy*. In LINNAEUS, an algorithm is specified by defining variables, parameters, functions, oracles, and update equations. All variables and parameters are symbolic, so there is no need to specialize problem dimensions or parameter choices. The system automatically translates an input algorithm into its state-space realization and computes the transfer function. The transfer functions can be compared and manipulated as needed to establish various kinds of equivalences or other relations between algorithms.

Parameter declaration. Parameters of the algorithm can be declared as scalar (commutative) or vector or matrix (noncommutative). The following code shows how to add scalar `t` and matrix `L` to `algo1`.

```
# add a scalar parameter t
t = algo1.add_parameter("t")
# add a matrix parameter L
L = algo1.add_parameter("L", commutative = False)
```

Parameter specification. Given two input algorithms, LINNAEUS computes the transfer functions and can compare them to detect equivalence and other relations. Some algorithms are equivalent or related only when the parameters satisfy a certain condition: for example, DR and ADMM. If the transfer functions of each algorithm use different parameters, LINNAEUS form symbolic equations and solve the equations to determine conditions that, if satisfied by the algorithm parameters, yield the desired relation between the algorithms; see (5.5) in section 5.

Oracles and function. Oracles play the starring role in our framework: oracle equivalence is possible only if two algorithms share the same oracles. In LINNAEUS, we provide two approaches to declare and add oracles to an algorithm. The black-box approach is to define oracles as black boxes. When parsing the algorithm, the system treats each oracle as a distinct entity unrelated to any other oracle. An oracle declared using syntax `add_oracle` uses the black-box approach. For example, we may add oracles ∇f and prox_g to algorithm `algo1`:

```
# add oracle gradient of f in the first approach
gradf = algo1.add_oracle("gradf")
# add oracle prox of g in the first approach
proxg = algo1.add_oracle("proxg")
```

The functional approach is to define oracles in terms of the (sub)gradient of a function. When parsing an algorithm, all the oracles will be decomposed into (sub)gradients and the state-space realization given in terms of (sub)gradients. We say that two algorithms are oracle-equivalent in terms of functional oracles if they are oracle-equivalent after rewriting the algorithm to use only (sub)gradient oracles. This approach is critical to allow us to identify algorithm conjugation, since conjugate algorithms use different (conjugate) oracles. If every algorithm is represented in terms of (sub)gradients, algorithm conjugation can be detected using proposition 8.1. Fortunately, common oracles such as prox and argmin can be easily written in terms of (sub)gradients: for example, $\text{prox}_f(x) = (I - \partial f)^{-1}(x)$ and argmin as (9.1).

To use the functional approach, users must define and add functions to the algorithm first using `add_function` and then declare and add oracles. The following code shows how to use the functional approach to declare and add oracles ∇f and prox_f.

```
# add function f
f = algo1.add_function("f")
# gradient of f with repect to x1
lin.grad(f)(x1)
# prox of f with repect to x2 and parameter t
lin.prox(f,t)(x2)
```

9.3. Black-box vs functional oracles. Are two algorithms equivalent with respect to black-box oracles if and only if they are equivalent with respect to functional oracles? Intuitively, when oracles are defined in terms of (sub)gradients, it might be possible to identify more relations with other algorithms. However, as stated in proposition 9.1, for algorithms that use only proximal operators, argmins, and (sub)gradients as oracles, equivalence is preserved under both black-box and functional definitions of oracles.

PROPOSITION 9.1. *Suppose two algorithms use only proximal operators, argmins, and (sub)gradients as oracles. Then the two algorithms are equivalent with respect to black-box oracles if and only if they are also equivalent with respect to functional oracles.*

Proof. Since for any function g and any t, $\operatorname{prox}_{tg}(x) = \operatorname{argmin}_y\{tg(y)+\frac{1}{2}\|x-y\|^2\}$, we can treat proximal operator as a special case of argmin. Without loss of generality, any argmin oracle in a linear algorithm has the form

$$z = \operatorname{argmin}_x \left\{ \lambda g(x) + \frac{1}{2} \begin{bmatrix} x \\ y \end{bmatrix}^T \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \right\}.$$

Here z is the value of the oracle and y can be regarded as the argument, which means from the perspective of a linear system, z is the input and y is the output. The parameter λ can be a scalar or matrix, g is a function, and Q_{11}, Q_{12}, Q_{21}, Q_{22} are parameter matrices. Specifically,

$$\begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix}$$

is a symmetric matrix and

$$\frac{1}{2} \begin{bmatrix} x \\ y \end{bmatrix}^T \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

is a quadratic term with respect to x and y. The matrix Q_{11} must be invertible if the argmin oracle is single-valued. To recover the proximal operator, choose a scalar λ and set

$$\begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} = \begin{bmatrix} I & -I \\ -I & I \end{bmatrix}.$$

If g is a convex function, the argmin oracle can be written in terms of the subgradient oracle ∂g as follows [2, §6; 13, §2],

$$z \in -Q_{11}^{-1}\lambda \partial g(z) - Q_{11}^{-1}Q_{12}y. \tag{9.1}$$

Suppose we have an algorithm with $n+m$ oracles in total, consisting of n argmins and m (sub)gradients. We can group the argmins and the (sub)gradients together respectively and partition the state-space realization accordingly as

$$\left[\begin{array}{c|cc} A & B_1 & B_2 \\ \hline C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \end{array}\right], \tag{9.2}$$

where C_1, B_1 correspond to the argmins, C_2, B_2 correspond to the (sub)gradients, and D is partitioned accordingly into D_{11}, D_{12}, D_{21}, and D_{22}. The transfer function can be represented accordingly as

$$\hat{H}(z) = \begin{bmatrix} \hat{H}_{11}(z) & \hat{H}_{12}(z) \\ \hat{H}_{21}(z) & \hat{H}_{22}(z) \end{bmatrix} = \begin{bmatrix} C_1(zI-A)^{-1}B_1 + D_{11} & C_1(zI-A)^{-1}B_2 + D_{12} \\ C_2(zI-A)^{-1}B_1 + D_{21} & C_2(zI-A)^{-1}B_2 + D_{22} \end{bmatrix}.$$

The input and output are partitioned as $(\bar{u}_1, \bar{u}_2)$ and $(\bar{y}_1, \bar{y}_2)$, where $\bar{y}_1 = (y_1, \ldots, y_n)$, $\bar{y}_2 = (y_{n+1}, \ldots, y_{n+m})$, $\bar{u}_1 = (z_1, \ldots, z_n)$, and $\bar{u}_2 = (\nabla f_{n+1}(y_{n+1}), \ldots, \nabla f_{n+m}(y_{n+m}))$. For each $i \in \{1, \ldots, n\}$ we have

$$z_i = \operatorname{argmin}_x \left\{ \lambda_i f_i(x) + \frac{1}{2} \begin{bmatrix} x \\ y_i \end{bmatrix}^T \begin{bmatrix} Q_{11}^i & Q_{12}^i \\ Q_{21}^i & Q_{22}^i \end{bmatrix} \begin{bmatrix} x \\ y_i \end{bmatrix} \right\} \tag{9.3}$$

where Q_{11}^i is invertible for any $i \in \{1, \ldots, n\}$.

Now we rewrite the linear system so that the nonlinearities corresponding to the argmins for the new linear system are (sub)gradients. Let $\lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_n)$, $Q_1 = \operatorname{diag}(Q_{11}^1, \ldots, Q_{11}^n)$, $Q_2 = \operatorname{diag}(Q_{12}^1, \ldots, Q_{12}^n)$, and $M_1 = Q_1^{-1} Q_2$, and $M_2 = Q_1^{-1} \lambda$. The new state-space realization in terms of the (sub)gradient oracles is

$$\left[\begin{array}{c|cc} A - B_1(I + M_1 D_{11})^{-1} M_1 C_1 & -B_1(I + M_1 D_{11})^{-1} M_2 & B_2 - B_1(I + M_1 D_{11})^{-1} M_1 D_{12} \\ \hline -(I + M_1 D_{11})^{-1} M_1 C_1 & -(I + M_1 D_{11})^{-1} M_2 & -(I + M_1 D_{11})^{-1} M_1 D_{12} \\ C_2 - D_{21}(I + M_1 D_{11})^{-1} M_1 C_1 & -D_{21}(I + M_1 D_{11})^{-1} M_2 & D_{22} - D_{21}(I + M_1 D_{11})^{-1} M_1 D_{12} \end{array}\right]. \tag{9.4}$$

We can compute the transfer function as

$$\hat{H}'(z) = \begin{bmatrix} \hat{H}'_{11}(z) & \hat{H}'_{12}(z) \\ \hat{H}'_{21}(z) & \hat{H}'_{22}(z) \end{bmatrix} = \begin{bmatrix} -(I + M_1 \hat{H}_{11}(z))^{-1} M_2 & -(I + M_1 \hat{H}_{11}(z))^{-1} M_1 \hat{H}_{12}(z) \\ -\hat{H}_{21}(z)(I + M_1 \hat{H}_{11}(z))^{-1} M_2 & \hat{H}_{22}(z) - \hat{H}_{21}(z)(I + M_1 \hat{H}_{11}(z))^{-1} M_1 \hat{H}_{12}(z) \end{bmatrix}. \tag{9.5}$$

Note that $I + M_1 D_{11}$ is invertible (otherwise the algorithm is not causal) and consequently $I + M_1 \hat{H}_{11}(z)$ is invertible. The matrix Q_1 is also invertible, since Q_{11}^i is invertible for any $i \in \{1, \ldots, n\}$. A detailed proof of (9.4) and (9.5) is provided in appendix H. Therefore, we know that if $\hat{H}(z)$ is fixed then $\hat{H}'(z)$ is also fixed. ∎

10. Conclusion and future work. 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, or algorithms that are the same up to shifts of the update equations, repetition of the updates with the same unit block, and conjugation of the function oracles. These ideas are implemented in the software package Linnaeus, which allows researchers to search for algorithms that are related to a given input and identify parameter settings that make the algorithms equivalent. Our goal is to allow researchers add new algorithms to Linnaeus as they are developed, so that Linnaeus can remain a valuable resource for algorithm designers seeking to understand connections (if any) to previous methods.

Our framework requires that the algorithm is linear in the state and oracle outputs, but not necessarily in the parameters. This constraint still allows us to handle a surprisingly large class of algorithms. There are several interesting directions for future work.

Can we detect equivalence between stochastic or randomized algorithms? Our framework applies to such algorithms with almost no modifications, simply by allowing random oracles. For example, we can accept oracles like random search $\operatorname{argmin}\{f(x + \omega_i) : i = 1, \ldots, k\}$, stochastic gradient $\nabla f(x) + \omega$, or noisy gradient $\nabla f(x + \omega)$. The definition of oracle equivalence would need a slight modification: 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.

Can we detect equivalence between parallel or distributed algorithms? Surprisingly, our framework still works for parallel or distributed algorithms. Notice that in a parallel algorithm, many oracle calls may be independently executed on different processors at about the same time. The precise ordering of these calls is not determined by the algorithm, and so different runs of the algorithm can generate different oracle sequences. However, all the possible oracle sequences generated by the same algorithm share the same dependence graph. Using the formalism defined in subsection 6.1, we can see that our framework can identify equivalence between parallel or distributed algorithms using the expanded definition of oracle equivalence: two algorithms are oracle-equivalent if there exists a way of writing each algorithm as a sequence of updates so that they generate identical sequences of oracle calls.

Can we detect equivalence between adaptive or nonlinear algorithms? Transfer functions are only defined for linear time-invariant (LTI) systems, so the LTI assumption in our framework is critical. Nevertheless, many of the other concepts from subsection 3.3 do extend to systems that are *almost* LTI. 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 [1], and the notion of a transfer function has been generalized to LTV systems [17]. If, instead, the parameters change adaptively based on the other state variables, the system can be regarded as a linear parameter varying (LPV) system [21] or a switched system [30]. Examples of such algorithms include nonlinear conjugate gradient methods and quasi-Newton methods.

For these more complicated cases, it is still reasonable to ask whether two algorithms invoke the same sequence of oracle calls. Discovering representations for nonlinear or time-varying algorithms that suffice to check equivalence is an interesting direction for future research.

REFERENCES

[1] P. J. Antsaklis and A. N. Michel, *Linear systems*, Birkhäuser, 2006.
[2] A. Beck, *First-order methods in optimization*, SIAM, 2017.
[3] A. Beck and M. Teboulle, *A fast iterative shrinkage-thresholding algorithm for linear inverse problems*, SIAM Journal on Imaging Sciences, 2 (2009), pp. 183–202.
[4] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, *Distributed optimization and statistical learning via the alternating direction method of multipliers*, Foundations and Trends in Machine Learning, 3 (2011), pp. 1–122.
[5] S. Boyd and L. Vandenberghe, *Convex optimization*, Cambridge University Press, 2004.
[6] S. Bubeck, *Convex optimization: algorithms and complexity*, Foundations and Trends® in Machine Learning, 8 (2015), pp. 231–357.
[7] Y. Censor, A. Gibali, and S. Reich, *The subgradient extragradient method for solving variational inequalities in hilbert space*, Journal of Optimization Theory and Applications, 148 (2011), pp. 318–335.
[8] A. Chambolle and T. Pock, *A first-order primal-dual algorithm for convex problems with applications to imaging*, Journal of mathematical imaging and vision, 40 (2011), pp. 120–145.
[9] C.-K. Chiang, T. Yang, C.-J. Lee, M. Mahdavi, C.-J. Lu, R. Jin, and S. Zhu, *Online optimization with gradual variations*, in Conference on Learning Theory, 2012, pp. 1–6.
[10] C. Daskalakis, A. Ilyas, V. Syrgkanis, and H. Zeng, *Training GANs with optimism*, in International Conference on Learning Representations, 2018.
[11] J. Douglas and H. H. Rachford, *On the numerical solution of heat conduction problems in two and three space variables*, Transactions of the American mathematical Society, 82 (1956), pp. 421–439.
[12] J. Eckstein and D. P. Bertsekas, *On the douglas-rachford splitting method and the proximal point algorithm for maximal monotone operators*, Mathematical Programming, 55 (1992), pp. 293–318.
[13] W. Fenchel, *Convex cones, sets and functions, mimeographed notes*, Princeton University, 1953.
[14] G. Gidel, H. Berard, G. Vignoud, P. Vincent, and S. Lacoste-Julien, *A variational inequality perspective on generative adversarial networks*, in International Conference on Learning Representations, 2019.
[15] J. H. Goodnight, *A tutorial on the sweep operator*, The American Statistician, 33 (1979), pp. 149–158.
[16] B. Hu, P. Seiler, and L. Lessard, *Analysis of biased stochastic gradient descent using sequential semidefinite programs*, Mathematical Programming, (2020), pp. 1–26.
[17] E. W. Kamen, P. P. Khargonekar, and K. R. Poolla, *A transfer-function approach to linear time-varying discrete-time systems*, SIAM Journal on Control and Optimization, 23 (1985), pp. 550–565, https://doi.org/10.1137/0323035.
[18] L. Lessard, B. Recht, and A. Packard, *Analysis and design of optimization algorithms via integral quadratic constraints*, SIAM Journal on Optimization, 26 (2016), pp. 57–95.
[19] P.-L. Lions and B. Mercier, *Splitting algorithms for the sum of two nonlinear operators*, SIAM Journal on Numerical Analysis, 16 (1979), pp. 964–979.
[20] Y. Malitsky, *Projected reflected gradient methods for monotone variational inequalities*, SIAM Journal on Optimization, 25 (2015), pp. 502–520.
[21] J. Mohammadpour and C. Scherer, *Control of linear parameter varying systems with applications*, Springer New York, 2012.
[22] J. J. Moreau, *Décomposition orthogonale d'un espace hilbertien selon deux cônes mutuellement polaires*, Comptes rendus hebdomadaires des séances de l'Académie des sciences, 255 (1962), pp. 238–240.
[23] Y. Nesterov, *Lectures on convex optimization*, Springer, 2018.
[24] D. O'Connor and L. Vandenberghe, *On the equivalence of the primal-dual hybrid gradient method and douglas–rachford splitting*, Mathematical Programming, 179 (2020), pp. 85–108.
[25] N. Parikh and S. Boyd, *Proximal algorithms*, Foundations and Trends in Optimization, 1 (2014), pp. 127–239.
[26] L. D. Popov, *A modification of the arrow-hurwicz method for search of saddle points*, Mathematical notes of the Academy of Sciences of the USSR, 28 (1980), pp. 845–848.
[27] A. Rakhlin and K. Sridharan, *Online learning with predictable sequences*, vol. 30 of Proceedings of Machine Learning Research, 2013, pp. 993–1019.
[28] E. K. Ryu and S. Boyd, *Primer on monotone operator methods*, Appl. Comput. Math, 15 (2016), pp. 3–43.
[29] E. K. Ryu and W. Yin, *Large-scale convex optimization via monotone operators*, Draft, 2020.

[30] Z. Sun, *Switched linear systems: control and design*, Springer Science & Business Media, 2006.
[31] M. J. Tsatsomeros, *Principal pivot transforms: properties and applications*, Linear Algebra and its Applications, 307 (2000), pp. 151–165.
[32] B. Van Scoy, R. A. Freeman, and K. M. Lynch, *The fastest known globally convergent first-order method for minimizing strongly convex functions*, IEEE Control Systems Letters, 2 (2017), pp. 49–54.
[33] F. Vasilyev, E. Khoroshilova, and A. Antipin, *An extragradient method for finding the saddle point in an optimal control problem*, Moscow University Computational Mathematics and Cybernetics, 34 (2010), pp. 113–118.
[34] Z. Wen, D. Goldfarb, and W. Yin, *Alternating direction augmented lagrangian methods for semidefinite programming*, Mathematical Programming Computation, 2 (2010), pp. 203–230.
[35] R. L. Williams, D. A. Lawrence, et al., *Linear state-space control systems*, John Wiley & Sons, 2007.

Appendix A. Proof of (3.14). Since $\mathbf{u}$ and $\mathbf{y}$ are partitioned as $\mathbf{u} = (\mathbf{u}_1, \mathbf{u}_2)$ and $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2)$, the state-space realization can be partitioned accordingly as

$$\left[\begin{array}{c|cc} A & B_1 & B_2 \\ \hline C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \end{array}\right].$$

We can express the transfer function $\hat{H}(z)$ as

$$\hat{H}(z) = \begin{bmatrix} \hat{H}_{11}(z) & \hat{H}_{12}(z) \\ \hat{H}_{21}(z) & \hat{H}_{22}(z) \end{bmatrix} = \begin{bmatrix} C_1(zI-A)^{-1}B_1 + D_{11} & C_1(zI-A)^{-1}B_2 + D_{12} \\ C_2(zI-A)^{-1}B_1 + D_{21} & C_2(zI-A)^{-1}B_2 + D_{22} \end{bmatrix}.$$

The system equations show as

$$\begin{aligned} x^{k+1} &= Ax^k + B_1u_1^k + B_2u_2^k \\ y_1^k &= C_1x^k + D_{11}u_1^k + D_{12}u_2^k \\ y_2^k &= C_2x^k + D_{21}u_1^k + D_{22}u_2^k. \end{aligned} \tag{A.1}$$

As we invert the input-output map corresponding to $\mathbf{y}_1$ and $\mathbf{u}_1$, the input of this system becomes (y_1^k, u_2^k) and the output is (u_1^k, y_2^k) at time k. From (A.1), as D_{11} is invertibe, we have

$$u_1^k = -D_{11}^{-1}C_1x^k + D_{11}^{-1}y_1^k - D_{11}^{-1}D_{12}u_2^k. \tag{A.2}$$

The new system equations change to

$$\begin{aligned} x^{k+1} &= (A - B_1D_{11}^{-1}C_1)x^k + B_1D_{11}^{-1}y_1^k + (B_2 - B_1D_{11}^{-1}D_{12})u_2^k \\ u_1^k &= -D_{11}^{-1}C_1x^k + D_{11}^{-1}y_1^k - D_{11}^{-1}D_{12}u_2^k \\ y_2^k &= (C_2 - D_{21}D_{11}^{-1}C_1)x^k + D_{21}D_{11}^{-1}y_1^k + (D_{22} - D_{21}D_{11}^{-1}D_{12})u_2^k, \end{aligned} \tag{A.3}$$

which correspond to state-space realization

$$\left[\begin{array}{c|cc} A - B_1D_{11}^{-1}C_1 & B_1D_{11}^{-1} & B_2 - B_1D_{11}^{-1}D_{12} \\ \hline -D_{11}^{-1}C_1 & D_{11}^{-1} & -D_{11}^{-1}D_{12} \\ (C_2 - D_{21}D_{11}^{-1}C_1) & D_{21}D_{11}^{-1} & D_{22} - D_{21}D_{11}^{-1}D_{12} \end{array}\right]. \tag{A.4}$$

To calculate the transfer function, note that

$$\begin{aligned} (zI - A + B_1D_{11}^{-1}C_1)^{-1} &= (zI-A)^{-1} + (zI-A)^{-1}B_1(-D_{11} - C_1(zI-A)^{-1}B_1)^{-1}C_1(zI-A)^{-1} \\ &= (zI-A)^{-1} - (zI-A)^{-1}B_1\hat{H}_{11}^{-1}(z)C_1(zI-A)^{-1}. \end{aligned}$$

We have

$$
\begin{aligned}
\hat{H}'_{11}(z) &= -D_{11}^{-1}C_1((zI-A)^{-1} - (zI-A)^{-1}B_1\hat{H}_{11}^{-1}(z)C_1(zI-A)^{-1})B_1D_{11}^{-1} + D_{11}^{-1} \\
&= -D_{11}^{-1}(\hat{H}_{11}(z) - D_{11} - (\hat{H}_{11}(z) - D_{11})\hat{H}_{11}^{-1}(z)(\hat{H}_{11}(z) - D_{11}))D_{11}^{-1} + D_{11}^{-1} \\
&= -D_{11}^{-1}(\hat{H}_{11}(z) - D_{11})(I - \hat{H}_{11}^{-1}(z)(\hat{H}_{11}(z) - D_{11}))D_{11}^{-1} + D_{11}^{-1} \\
&= -D_{11}^{-1}(\hat{H}_{11}(z) - D_{11})\hat{H}_{11}^{-1}(z) + D_{11}^{-1} \\
&= \hat{H}_{11}^{-1}(z) \\
\hat{H}'_{12}(z) &= -D_{11}^{-1}C_1((zI-A)^{-1} - (zI-A)^{-1}B_1\hat{H}_{11}^{-1}(z)C_1(zI-A)^{-1})B_2 - \hat{H}_{11}^{-1}(z)D_{12} \\
&= -D_{11}^{-1}(\hat{H}_{12}(z) - D_{12} - (\hat{H}_{11}(z) - D_{11})\hat{H}_{11}^{-1}(z)(\hat{H}_{12}(z) - D_{12})) - \hat{H}_{11}^{-1}(z)D_{12} \\
&= -D_{11}^{-1}(I - (\hat{H}_{11}(z) - D_{11})\hat{H}_{11}^{-1}(z))(\hat{H}_{12}(z) - D_{12}) - \hat{H}_{11}^{-1}(z)D_{12} \\
&= -\hat{H}_{11}^{-1}(z)(\hat{H}_{12}(z) - D_{12}) - \hat{H}_{11}^{-1}(z)D_{12} \\
&= -\hat{H}_{11}^{-1}(z)\hat{H}_{12}(z) \\
\hat{H}'_{21}(z) &= C_2((zI-A)^{-1} - (zI-A)^{-1}B_1\hat{H}_{11}^{-1}(z)C_1(zI-A)^{-1})B_1D_{11}^{-1} + D_{21}\hat{H}_{11}^{-1}(z) \\
&= (\hat{H}_{21}(z) - D_{21} - (\hat{H}_{21}(z) - D_{21})\hat{H}_{11}^{-1}(z)(\hat{H}_{11}(z) - D_{11}))D_{11}^{-1} + D_{21}\hat{H}_{11}^{-1}(z) \\
&= (\hat{H}_{21}(z) - D_{21})(I - \hat{H}_{11}^{-1}(z)(\hat{H}_{11}(z) - D_{11}))D_{11}^{-1} + D_{21}\hat{H}_{11}^{-1}(z) \\
&= (\hat{H}_{21}(z) - D_{21})\hat{H}_{11}^{-1}(z) + D_{21}\hat{H}_{11}^{-1}(z) \\
&= \hat{H}_{21}(z)\hat{H}_{11}^{-1}(z) \\
\hat{H}'_{22}(z) &= \hat{H}_{22}(z) - (\hat{H}_{21}(z) - D_{21})\hat{H}_{11}^{-1}(z)(\hat{H}_{12}(z) - D_{12}) - D_{21}\hat{H}_{11}^{-1}(z)(\hat{H}_{12}(z) - D_{12}) \\
&\quad - (\hat{H}_{21}(z) - D_{21})\hat{H}_{11}^{-1}(z)D_{12} - D_{21}\hat{H}_{11}^{-1}(z)D_{12} \\
&= \hat{H}_{22}(z) - \hat{H}_{21}(z)\hat{H}_{11}^{-1}(z)\hat{H}_{12}(z).
\end{aligned}
$$

Thus, we get the desired results as (3.14).

Appendix B. Proof of proposition 6.1.

Algorithm B.1 General form of algorithm $\mathcal{A}$

for $k = 0, 1, 2, \ldots$ **do**

$$
\begin{aligned}
x_1^{k+1} &= L_1(x_1^k, \ldots, x_m^k) \\
x_2^{k+1} &= L_2(x_1^{k+1}, x_2^k, \ldots, x_m^k) \\
&\vdots \\
x_i^{k+1} &= L_i(x_1^{k+1}, \ldots, x_{i-1}^{k+1}, x_i^k, \ldots, x_m^k, u_1^{k+1}) \\
&\vdots \\
x_{\tilde{i}}^{k+1} &= L_{\tilde{i}}(x_1^{k+1}, \ldots, x_{\tilde{i}-1}^{k+1}, x_{\tilde{i}}^k, \ldots, x_m^k, u_n^{k+1}) \\
&\vdots \\
x_m^{k+1} &= L_m(x_1^{k+1}, \ldots, x_{m-1}^{k+1}, x_m^k)
\end{aligned}
$$

end for

Without loss of generality, we can express algorithm $\mathcal{A}$ in the general form as algorithm B.1. Since $\mathcal{A}$ is an linear algorithm, $L_1, \ldots, L_m$ are linear functions. Given an initialization $\{x_1^0, \ldots, x_m^0\}$, $\mathcal{A}$ generates state sequence $(x_1^k, \ldots, x_m^k)_{k\geq 0}$, input sequence $(u_1^k, \ldots, u_n^k)_{k\geq 1}$, and output sequence $(y_1^k, \ldots, y_n^k)_{k\geq 1}$. The ith update equation is the first update equation that contains an oracle call, corresponding to u_1^k and y_1^k. The $\tilde{i}$th update equation is the last update equation that contains an oracle call, corresponding to u_n^k and

y_n^k. The outputs are also linear functions of the states. Specifically, we have

$$\begin{aligned} y_1^k =& Y_1(x_1^{k+1},\dots,x_{i-1}^{k+1},x_i^k,\dots,x_m^k),\quad u_1^k=\phi_1(y_1^k)\\ &\vdots\\ y_n^k =& Y_n(x_1^{k+1},\dots,x_{\tilde{i}-1}^{k+1},x_{\tilde{i}}^k,\dots,x_m^k),\quad u_n^k=\phi_n(y_n^k). \end{aligned}$$

Functions $Y_1,\dots,Y_n$ are linear functions and $\phi_1,\dots,\phi_n$ denote the oracle calls. Without loss of generality, suppose permutation $\tilde{\pi}=(\tilde{l}+1,\dots,m,1,\dots,\tilde{l})$ with $1<\tilde{l}<m$.

First case. Suppose the new order of oracle calls within one iteration is a cyclic permutation π of (n) (not identical to (n)). Without loss of generality, suppose $\pi=(j+1,\dots,n,1,\dots,j)$ with $1<j<n$, the jth oracle call corresponds to the $\tilde{j}$th update equation, and the $j+1$th oracle call corresponds to the $\tilde{p}$th update equation. By definition, we have $i\le\tilde{j}<\tilde{l}+1\le\tilde{p}\le\tilde{i}$. At the first time step $k=1$, the first input is u_1^1 and the first output is y_1^1, and the $j+1$th input and output are u_{j+1}^1 and y_{j+1}^1. We have

$$\begin{aligned} y_1^1 &= Y_1(x_1^1,\dots,x_{i-1}^1,x_i^0,\dots,x_m^0)\\ x_{\tilde{l}+1}^1 &= L_{\tilde{l}+1}(x_1^1,\dots,x_{\tilde{l}}^1,x_{\tilde{l}+1}^0,\dots,x_m^0)\\ y_{j+1}^1 &= Y_1(x_1^1,\dots,x_{\tilde{p}-1}^1,x_{\tilde{p}}^0,\dots,x_m^0). \end{aligned}$$

Here without loss of generality, suppose the $\tilde{l}+1$th update equation does not contain an oracle call. In other words, $\tilde{j}<\tilde{l}+1<\tilde{p}$. By definition, $\mathcal{B}$ calls the update equations in the order $\tilde{\pi}$. At the first time step, the $\tilde{l}+1$th update equation is first called. If $\mathcal{B}$ is suitably initialized with states $\{x_1^1,\dots,x_{\tilde{l}}^1,x_{\tilde{l}+1}^0,\dots,x_m^0\}$, it will generate state sequence $(x_{\tilde{l}+1}^k,\dots,x_m^k,x_1^{k+1},\dots,x_{\tilde{l}}^{k+1})_{k\ge0}$, input sequence $(u_{j+1}^k,\dots,u_n^k,u_1^{k+1},\dots,u_j^{k+1})_{k\ge1}$, and output sequence $(y_{j+1}^k,\dots,y_n^k,y_1^{k+1},\dots,y_j^{k+1})_{k\ge1}$. The input and output sequences of $\mathcal{A}$ and $\mathcal{B}$ match up to prefixes $(u_1^1,\dots,u_j^1)$ and $(y_1^1,\dots,y_j^1)$ respectively. Therefore, $\mathcal{A}$ and $\mathcal{B}$ are shift-equivalent.

Second case. Suppose the order of oracle calls within one iteration remain unchanged (identical to (n)). By definition, we have $1<\tilde{l}+1\le i$. We have

$$\begin{aligned} x_{\tilde{l}+1}^1 &= L_{\tilde{l}+1}(x_1^1,\dots,x_{\tilde{l}}^1,x_{\tilde{l}+1}^0,\dots,x_m^0)\\ y_1^1 &= Y_1(x_1^1,\dots,x_{i-1}^1,x_i^0,\dots,x_m^0). \end{aligned}$$

Here without loss of generality, suppose the $\tilde{l}+1$th update equation does not contain an oracle call. In other words, $\tilde{l}+1<i$. By definition, $\mathcal{B}$ calls the update equations in the order $\tilde{\pi}$. At the first time step, the $\tilde{l}+1$th update equation is first called. If $\mathcal{B}$ is suitably initialized with states $\{x_1^1,\dots,x_{\tilde{l}}^1,x_{\tilde{l}+1}^0,\dots,x_m^0\}$, it will generate state sequence $(x_{\tilde{l}+1}^k,\dots,x_m^k,x_1^{k+1},\dots,x_{\tilde{l}}^{k+1})_{k\ge0}$. The input and output sequences remain unchanged. Therefore, $\mathcal{A}$ and $\mathcal{B}$ are oracle-equivalent. Meanwhile, since oracle equivalence can be regarded as a special case of shift equivalence, $\mathcal{A}$ and $\mathcal{B}$ are also shift-equivalent.

Appendix C. Discussions on permutation and its generalization. To take a revisit of proposition 6.3, it can be found that as algorithm $\mathcal{A}$ is permuted to make the order of oracle calls within one iteration as $(j+1,\dots,n,1,\dots,j)$ from $(1,\dots,n)$, the resulting transfer function is exactly the same as adding a one-step time delay to channels (oracles) $(1,\dots,j)$ according to results in control theory. Another interpretation of adding a one-step time delay comes from the system equations (6.5). We can see that the input and output corresponding to channels (oracles) $(1,\dots,j)$ are the input and output for the next time step $\bar{u}_1^{k+1}$ and $\bar{y}_1^{k+1}$, however, the input and output of channels (oracles) $(j+1,\dots,n)$ are still the ones for the current time step $\bar{u}_2^k$ and $\bar{y}_2^k$. Intrinsically, after cyclic permutation, the intrinsic update order of oracles does not change, but a one-step time delay is added to the oracles that we would like to update latterly.

Using the idea of time delay, we can generalize algorithm permutation as adding any step of time delay to any channel (oracle) of an algorithm. Suppose we add time delay to oracle i of algorithm $\mathcal{A}$ by d_i for any $i\in(n)$, where d_i can be any integer, the resulting algorithm $\mathcal{B}$ has transfer function $\hat{H}_{\mathcal{B}}(z)$ as

$$\hat{H}_{\mathcal{B}}(z)=\begin{bmatrix} z^{d_1} & & & \\ & z^{d_2} & & \\ & & \ddots & \\ & & & z^{d_n} \end{bmatrix}\hat{H}_{\mathcal{A}}(z)\begin{bmatrix} z^{-d_1} & & & \\ & z^{-d_2} & & \\ & & \ddots & \\ & & & z^{-d_n} \end{bmatrix}, \tag{C.1}$$

where $\hat{H}_{\mathcal{A}}(z)$ is the transfer function of $\mathcal{A}$.

To be more specific, suppose we add time delay d_i to oracle i for algorithm $\mathcal{A}$, $h_{\mathcal{A}}^{kl}(z)$ with $1 \le k \le n$ and $1 \le l \le n$ denotes the entry of $\hat{H}_{\mathcal{A}}(z)$. The transfer function of the resulting algorithm $\mathcal{B}$ can be expressed entrywise as

$$h_{\mathcal{B}}^{kl}(z) = \begin{cases} h_{\mathcal{A}}^{ii}(z) & k = i \ l = i \\ h_{\mathcal{A}}^{il}(z) z^{d_i} & k = i \ l \neq i \\ h_{\mathcal{A}}^{ki}(z) z^{-d_i} & k \neq i \ l = i \\ h_{\mathcal{A}}^{kl}(z) & k \neq i \ l \neq i \end{cases} \tag{C.2}$$

In this way, we know that proposition 6.3 is a special case of (C.1) with $d_1 = \cdots = d_j = 1$.

However, there are restrictions so that we cannot add any arbitrary step of time delay to any oracle. From section 3.3, transfer functions are rational (matrix) functions with respect to z. Further, the rational functions must be proper in order to make the transfer function realizable. From (C.2), as we add time delay d_i to oracle i for $\mathcal{A}$, the off-diagonal entries in the ith row of $\hat{H}_{\mathcal{A}}(z)$ are multiplied by z^{d_i} and the off-diagonal entries in the ith column of $\hat{H}_{\mathcal{A}}(z)$ are multiplied by z^{-d_i} while the ith diagonal entry remains unchanged. From the perspective of relative degrees, as relative degree is the difference between the degree of denominator and the degree of numerator, the relative degrees of the off-diagonal entries in the ith row are decreased by d_i but the relative degrees of the off-diagonal entries in the ith column are increased by d_i. Suppose the smallest relative degree among the off-diagonal entries in the ith row is r_i, then d_i must satisfy $d_i \le r_i$ to maintain properness of the resulting off-diagonal entries in the ith row. Similarly, suppose the smallest relative degree among the off-diagonal entries of the ith column is c_i, then d_i must satisfy $-d_i \le c_i$ to maintain properness of the resulting off-diagonal entries in the ith column. In other words, we can add time delay d_i to oracle i only if $-c_i \le d_i \le r_i$. Otherwise, at least one off-diagonal entry in the ith row or the ith column is no longer proper, leading to an invalid transfer function.

For any algorithm with state-space realization (A, B, C, D), the transfer function is calculated by $C(zI - A)^{-1}B + D$. Term $C(zI - A)^{-1}B$ is a strictly proper (matrix) function, where strictly proper means that the degree of z in the numerator polynomial is strictly less than the degree of z in the denominator polynomial. Thus, for any nonzero entry of D, the corresponding entry in the transfer function has relative degree zero. Take a revisit of cyclic permutation, for any causal algorithm, the entries above diagonal of the D matrix must be zero, especially after necessary reordering. Thus, the entries above diagonal in the transfer function have strictly positive relative degrees. This implies that any cyclic permutation of an algorithm always exists. Note that before performing cyclic permutation, we are required to reorder the state-space realization if needed.

Reconsider algorithms 6.5 and 6.6, in (6.11) and (6.12), comparing $\hat{H}_{10}(z)$ to $\hat{H}_{11}(z)$, we add a one-step time delay to the first channel. Term $\frac{1}{z-1}$ in $\hat{H}_{10}(z)$ is multiplied by z and term $\frac{2z-1}{z-1}$ is multiplied by z^{-1}. Further, the off-diagonal entry in the first row of $\hat{H}_{10}(z)$ has relative degree 1 and the off-diagonal entry in the first column of $\hat{H}_{10}(z)$ has relative degree 0. Thus, we can only add time delay $d_1 = 1$ to the first oracle of algorithm 6.5, as $0 \le d_1 \le 1$ to maintain properness.

Appendix D. Proof of shift-equivalence of DR and ADMM continued. Suppose the oracles for both DR (algorithm 6.5) and ADMM (algorithm 6.6) are subgradients of f and g. Oracles prox and argmin can be expanded as inclusions involving subgradients. The update equations of DR and ADMM can be rewritten into formations of algorithms D.1 and D.2 respectively. Note that the update equations involving subgradients are inclusions.

Algorithm D.1 DR

for $k = 0, 1, 2, \ldots$ **do**
$x_1^{k+1} \in x_3^k - t\partial f(x_1^{k+1})$
$x_2^{k+1} \in 2x_1^{k+1} - x_3^k - t\partial g(x_2^{k+1})$
$x_3^{k+1} = x_3^k + x_2^{k+1} - x_1^{k+1}$
end for

Algorithm D.2 ADMM

for $k = 0, 1, 2, \ldots$ **do**
$\xi_1^{k+1} \in \xi_2^k - \xi_3^k - \frac{1}{\rho}\partial g(\xi_1^{k+1})$
$\xi_2^{k+1} \in \xi_1^{k+1} + \xi_3^k - \frac{1}{\rho}\partial f(\xi_2^{k+1})$
$\xi_3^{k+1} = \xi_3^k + \xi_1^{k+1} - \xi_2^{k+1}$
end for

We still assume $\rho = 1/t$ in ADMM. The transfer functions are computed as $\hat{H}_{18}(z)$ and $\hat{H}_{19}(z)$ respec-

tively. Note that $\hat{H}_{19}(z)$ is not written in the causal order.

$$\hat{H}_{18}(z) = \left[\begin{array}{ccc|cc} 0 & 0 & 1 & -t & 0 \\ 0 & 0 & 1 & -2t & -t \\ 0 & 0 & 1 & -t & -t \\ \hline 0 & 0 & 1 & -t & 0 \\ 0 & 0 & 1 & -2t & -t \end{array}\right] = \left[\begin{array}{cc} -\frac{tz}{z-1} & -\frac{t}{z-1} \\ \frac{t-2tz}{z-1} & -\frac{tz}{z-1} \end{array}\right]$$

$$\hat{H}_{19}(z) = \left[\begin{array}{ccc|cc} 0 & 1 & -1 & 0 & -t \\ 0 & 1 & 0 & -t & -t \\ 0 & 0 & 0 & t & 0 \\ \hline 0 & 1 & 0 & -t & -t \\ 0 & 1 & -1 & 0 & -t \end{array}\right] = \left[\begin{array}{cc} -\frac{tz}{z-1} & -\frac{tz}{z-1} \\ \frac{t-2tz}{z(z-1)} & -\frac{tz}{z-1} \end{array}\right]$$

From propositions 6.1 and 6.3, we know that they are still (cyclic) permutation and they are shift-equivalent.

Appendix E. Proof of proposition 7.3. Suppose the oracles of algorithm $\mathcal{A} : \mathcal{X} \to \mathcal{X}$ can be represented as $\phi : \mathcal{X} \to \mathcal{X}$. Since $\mathcal{A}$ converges to fixed point $(y^\star, u^\star, x^\star)$, it satisfies

$$\begin{aligned} x^\star &= Ax^\star + Bu^\star \\ y^\star &= Cx^\star + Du^\star \\ u^\star &= \phi(y^\star). \end{aligned}$$

Therefore, we have

$$\begin{aligned} x^\star &= Ax^\star + Bu^\star \\ &= A(Ax^\star + Bu^\star) + Bu^\star \\ &= A^2x^\star + ABu^\star + Bu^\star \\ &= \dots \\ &= A^{n-1}x^\star + A^{n-2}Bu^\star + \cdots + ABu^\star + Bu^\star \\ &= A^nx^\star + A^{n-1}Bu^\star + \cdots + ABu^\star + Bu^\star \\ y^\star &= Cx^\star + Du^\star \\ &= C(Ax^\star + Bu^\star) + Du^\star \\ &= CAx^\star + CBu^\star + Du^\star \\ &= \dots \\ &= CA^{n-1}x^\star + CA^{n-2}Bu^\star + \cdots + CBu^\star + Du^\star. \end{aligned}$$

With (7.5), we have

$$\begin{aligned} x^\star &= A^nx^\star + A^{n-1}Bu^\star + \cdots + ABu^\star + Bu^\star \\ y^\star &= Cx^\star + Du^\star \\ y^\star &= CAx^\star + CBu^\star + Du^\star \\ &\vdots \\ y^\star &= CA^{n-1}x^\star + CA^{n-2}Bu^\star + \dots CBu^\star + Du^\star, \end{aligned}$$

which indicates that $\mathcal{A}^n$ converges to fixed point $(y', u', x^\star)$ with $y' = y^\star \bigotimes \mathbb{1}^n$ and $u' = u^\star \bigotimes \mathbb{1}^n$.

Appendix F. Proof of proposition 8.3. Without loss of generality, let the permutation matrix equal to the identity as proposition 8.1. To simplify the notations, let

$$\left[\begin{array}{c|cc} A & B[[n],\kappa] & B[[n],\bar{\kappa}] \\ \hline C[\kappa,[n]] & D[\kappa] & D[\kappa,\bar{\kappa}] \\ C[\bar{\kappa},[n]] & D[\bar{\kappa},\kappa] & D[\bar{\kappa}] \end{array}\right] = \left[\begin{array}{c|cc} A & B_1 & B_2 \\ \hline C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \end{array}\right],$$

$$\begin{bmatrix} \hat{H}[\kappa](z) & \hat{H}[\kappa,\bar{\kappa}](z) \\ \hat{H}[\bar{\kappa},\kappa](z) & \hat{H}[\bar{\kappa}](z) \end{bmatrix} = \begin{bmatrix} \hat{H}_{11}(z) & \hat{H}_{12}(z) \\ \hat{H}_{21}(z) & \hat{H}_{22}(z) \end{bmatrix} = \begin{bmatrix} C_1(zI-A)^{-1}B_1 + D_{11} & C_1(zI-A)^{-1}B_2 + D_{12} \\ C_2(zI-A)^{-1}B_1 + D_{21} & C_2(zI-A)^{-1}B_2 + D_{22} \end{bmatrix}.$$

In this way, $(y[\kappa]^\star, y[\bar{\kappa}]^\star, u[\kappa]^\star, u[\bar{\kappa}]^\star, x^\star)$ can be written as $(y_1^\star, y_2^\star, u_1^\star, u_2^\star, x^\star)$, and $(u[\kappa]^\star, y[\bar{\kappa}]^\star, y[\kappa]^\star, u[\bar{\kappa}]^\star, x^\star)$ can be written as $(u_1^\star, y_2^\star, y_1^\star, u_2^\star, x^\star)$.

Partition the oracle calls of algorithm $\mathcal{A} : \mathcal{X} \to \mathcal{X}$ into two nonlinear oracles ϕ_1 and ϕ_2. Oracle ϕ_1 corresponds to the oracle calls in set κ, and ϕ_2 corresponds to the remaining oracle calls. Since $\mathcal{A}$ converges to fixed point $(y_1^\star, y_2^\star, u_1^\star, u_2^\star, x^\star)$, it satisfies

$$\begin{aligned} x^\star &= Ax^\star + B_1u_1^\star + B_2u_2^\star \\ y_1^\star &= C_1x^\star + D_{11}u_1^\star + D_{12}u_2^\star \\ y_2^\star &= C_2x^\star + D_{21}u_1^\star + D_{22}u_2^\star \\ u_1^\star &= \phi_1(y_1^\star) \\ u_2^\star &= \phi_2(y_2^\star). \end{aligned}$$

The state-space realization of $\mathcal{B}$ is the same as (A.4). Note that D_{11} is invertible, we have

$$\begin{aligned} x^\star &= Ax^\star + B_2u_2^\star + B_1u_1^\star \\ &= Ax^\star + B_2u_2^\star + B_1(-D_{11}^{-1}C_1x^\star + D_{11}^{-1}y_1^\star - D_{11}^{-1}D_{12}u_2^\star) \\ &= (A - B_1D_{11}^{-1}C_1)x^\star + B_1D_{11}^{-1}y_1^\star + (B_2 - B_1D_{11}^{-1}D_{12})u_2^\star \\ u_1^\star &= -D_{11}^{-1}C_1x^\star + D_{11}^{-1}y_1^\star - D_{11}^{-1}D_{12}u_2^\star \\ y_2^\star &= C_2x^\star + D_{22}u_2^\star + D_{21}u_2^\star \\ &= C_2x^\star + D_{22}u_2^\star + D_{21}(D_{11}^{-1}y_1^\star - D_{11}^{-1}C_1x^\star - D_{11}^{-1}D_{12}u_2^\star) \\ &= (C_2 - D_{21}D_{11}^{-1}C_1)x^\star + D_{21}D_{11}^{-1}y_1^\star + (D_{22} - D_{21}D_{11}^{-1}D_{12})u_2^\star \\ y_1^\star &= \phi_1^{-1}(u_1^\star) \\ u_2^\star &= \phi_2(y_2^\star). \end{aligned}$$

Oracle ϕ_1^{-1} is the inverse oracle of oracle ϕ_1. Therefore, we get the desired results that algorithm $\mathcal{B}$ converges to fixed point $(u_1^\star, y_2^\star, y_1^\star, u_2^\star, x^\star)$.

Appendix G. Commutativity between conjugation and cyclic permutation.

PROPOSITION G.1. *Conjugation and cyclic permutation commute.*

Proof. Given an algorithm $\mathcal{A}$ with transfer function $\hat{H}(z)$. Suppose κ is a subset of the oracles of $\mathcal{A}$, D_κ is invertible, and $\pi = (m+1, \ldots, n, 1, \ldots, m)$ is an arbitrary cyclic permutation of the oracles of $\mathcal{A}$. We will show that the transfer functions of $\mathcal{C}_\kappa P_\pi \mathcal{A}$ and $P_\pi \mathcal{C}_\kappa \mathcal{A}$ are identical.

Suppose $\hat{H}^\star(z)$ is the transfer function of $P_\pi\mathcal{A}$, the results in proposition 6.3 can be written as

$$\hat{H}^\star(z) = Q\hat{H}(z)Q^{-1}. \tag{G.1}$$

Here Q is a diagonal matrix where the first m diagonal entries are all z and the rest of the diagonal entries are all ones. We will use the same settings and notations as proposition 8.1 to express changes in transfer function of conjugation $\mathcal{C}_\kappa$. Without loss of generality, the transfer function $\hat{H}'(z)$ of $\mathcal{C}_\kappa\mathcal{A}$ satisfies

$$\hat{H}'(z) = \begin{bmatrix} \hat{H}_{11}^{-1}(z) & -\hat{H}_{11}^{-1}(z)\hat{H}_{12}(z) \\ \hat{H}_{21}(z)\hat{H}_{11}^{-1}(z) & \hat{H}_{22}(z) - \hat{H}_{21}(z)\hat{H}_{11}^{-1}(z)\hat{H}_{12}(z) \end{bmatrix}. \tag{G.2}$$

Thus we can partition matrix Q as $\mathrm{diag}(Q_1, Q_2)$, where Q_1 corresponds to the oracles in κ and Q_2 corresponds to the rest part of oracles. Consequently, Q^{-1} can be written as $\mathrm{diag}({Q_1}^{-1}, {Q_2}^{-1})$.

From (G.1) and (G.2), we have

$$\hat{H}(z) \xrightarrow{\mathcal{C}_\kappa} \begin{bmatrix} \hat{H}_{11}^{-1}(z) & -\hat{H}_{11}^{-1}(z)\hat{H}_{12}(z) \\ \hat{H}_{21}(z)\hat{H}_{11}^{-1}(z) & \hat{H}_{22}(z) - \hat{H}_{21}(z)\hat{H}_{11}^{-1}(z)\hat{H}_{12}(z) \end{bmatrix}$$
$$\xrightarrow{P_\pi} \begin{bmatrix} Q_1\hat{H}_{11}^{-1}(z)Q_1^{-1} & -Q_1\hat{H}_{11}^{-1}(z)\hat{H}_{12}(z)Q_2^{-1} \\ Q_2\hat{H}_{21}(z)\hat{H}_{11}^{-1}(z)Q_1^{-1} & Q_2\hat{H}_{22}(z)Q_2^{-1} - Q_2\hat{H}_{21}(z)\hat{H}_{11}^{-1}(z)\hat{H}_{12}(z)Q_2^{-1} \end{bmatrix},$$
$$\hat{H}(z) \xrightarrow{P_\pi} \begin{bmatrix} Q_1\hat{H}_{11}(z)Q_1^{-1} & Q_1\hat{H}_{12}(z)Q_2^{-1} \\ Q_2\hat{H}_{21}(z)Q_1^{-1} & Q_2\hat{H}_{22}(z)Q_2^{-1} \end{bmatrix}$$
$$\xrightarrow{\mathcal{C}_\kappa} \begin{bmatrix} Q_1\hat{H}_{11}^{-1}(z)Q_1^{-1} & -Q_1\hat{H}_{11}^{-1}(z)\hat{H}_{12}(z)Q_2^{-1} \\ Q_2\hat{H}_{21}(z)\hat{H}_{11}^{-1}(z)Q_1^{-1} & Q_2\hat{H}_{22}(z)Q_2^{-1} - Q_2\hat{H}_{21}(z)\hat{H}_{11}^{-1}(z)\hat{H}_{12}(z)Q_2^{-1} \end{bmatrix}.$$

We get the desired results to show $\mathcal{C}_\kappa$ and P_π commute. Therefore, conjugation and cyclic permutation commute. ∎

Appendix H. Proof of (9.4) and (9.5). For each $i \in \{1, \dots, n\}$ we have

$$z_i = \operatorname{argmin}_x \left\{ \lambda_i f_i(x) + \frac{1}{2} \begin{bmatrix} x \\ y_i \end{bmatrix}^T \begin{bmatrix} Q_{11}^i & Q_{12}^i \\ Q_{21}^i & Q_{22}^i \end{bmatrix} \begin{bmatrix} x \\ y_i \end{bmatrix} \right\}.$$

Besides, Q_{11}^i is invertible for any $i \in \{1, \dots, n\}$. Since f_i is a convex function, the argmin oracle can be written as $z_i \in -{Q_{11}^i}^{-1}\lambda\partial f_i(z_i) - {Q_{11}^i}^{-1}Q_{12}^i y_i$ by treating ∂f_i as the oracle. Written into matrix form, we have

$$\bar{u}_1 = -{Q_1}^{-1}\lambda\tilde{u}_1 - {Q_1}^{-1}Q_2\bar{y}_1, \tag{H.1}$$

where $\tilde{u}_1 = [\partial f_1(z_1), \dots, \partial f_n(z_n)]^T$. Combine (H.1) with the state-space realization (9.2), we get the desired results for (9.4). The corresponding system equations show as

$$\begin{aligned} x^{k+1} &= (A - B_1(I + M_1D_{11})^{-1}M_1C_1)x^k - B_1(I + M_1D_{11})^{-1}M_2\tilde{u}_1^k + (B_2 - B_1(I + M_1D_{11})^{-1}M_1D_{12})\bar{u}_2^k \\ \bar{u}_1^k &= -(I + M_1D_{11})^{-1}M_1C_1x^k - (I + M_1D_{11})^{-1}M_2\tilde{u}_1^k - (I + M_1D_{11})^{-1}M_1D_{12}\bar{u}_2^k \\ \bar{y}_2^k &= (C_2 - D_{21}(I + M_1D_{11})^{-1}M_1C_1)x^k - D_{21}(I + M_1D_{11})^{-1}M_2\tilde{u}_1^k + (D_{22} - D_{21}(I + M_1D_{11})^{-1}M_1D_{12})\bar{u}_2^k. \end{aligned}$$

To calculate the transfer function, note that

$$(zI - A + B_1(I + M_1D_{11})^{-1}M_1C_1)^{-1} = (zI - A)^{-1} - (zI - A)^{-1}B_1(I + M_1\hat{H}_{11}(z))^{-1}M_1C_1(zI - A)^{-1}.$$

We have

$$\begin{aligned}
\hat{H}'_{11}(z) &= (I + M_1D_{11})^{-1}M_1C_1(zI - A + B_1(I + M_1D_{11})^{-1}M_1C_1)^{-1}B_1(I + M_1D_{11})^{-1}M_2 - (I + M_1D_{11})^{-1}M_2 \\
&= (I + M_1D_{11})^{-1}(M_1\hat{H}_{11}(z) - M_1D_{11})(I - (I + M_1\hat{H}_{11}(z))^{-1}(M_1\hat{H}_{11}(z) - M_1D_{11}))(I + M_1D_{11})^{-1}M_2 \\
&\quad - (I + M_1D_{11})^{-1}M_2 \\
&= (I + M_1D_{11})^{-1}(M_1\hat{H}_{11}(z) - M_1D_{11})(I + M_1\hat{H}_{11}(z))^{-1}M_2 - (I + M_1D_{11})^{-1}M_2 \\
&= -(I + M_1\hat{H}_{11}(z))^{-1}M_2 \\
\hat{H}'_{12}(z) &= -(I + M_1D_{11})^{-1}M_1C_1(zI - A + B_1(I + M_1D_{11})^{-1}M_1C_1)^{-1}B_2 - (I + M_1\hat{H}_{11}(z))^{-1}M_1D_{12} \\
&= -(I + M_1D_{11})^{-1}(I - (M_1\hat{H}_{11}(z) - M_1D_{11})(I + M_1\hat{H}_{11}(z))^{-1})(M_1\hat{H}_{12}(z) \\
&\quad - M_1D_{12}) - (I + M_1\hat{H}_{11}(z))^{-1}M_1D_{12} \\
&= -(I + M_1\hat{H}_{11}(z))^{-1}(M_1\hat{H}_{12}(z) - M_1D_{12}) - (I + M_1\hat{H}_{11}(z))^{-1}M_1D_{12} \\
&= -(I + M_1\hat{H}_{11}(z))^{-1}M_1\hat{H}_{12}(z) \\
\hat{H}'_{21}(z) &= -C_2(zI - A + B_1(I + M_1D_{11})^{-1}M_1C_1)^{-1}B_1(I + M_1D_{11})^{-1}M_2 - D_{21}(I + M_1\hat{H}_{11}(z))^{-1}M_2 \\
&= -(\hat{H}_{21}(z) - D_{21})(I - (I + M_1\hat{H}_{11}(z))^{-1}(M_1\hat{H}_{11}(z) - M_1D_{11}))(I + M_1D_{11})^{-1}M_2 \\
&\quad - D_{21}(I + M_1\hat{H}_{11}(z))^{-1}M_2 \\
&= -(\hat{H}_{21}(z) - D_{21})(I + M_1\hat{H}_{11}(z))^{-1}M_2 - D_{21}(I + M_1\hat{H}_{11}(z))^{-1}M_2 \\
&= -\hat{H}_{21}(z)(I + M_1\hat{H}_{11}(z))^{-1}M_2 \\
\hat{H}'_{22}(z) &= \hat{H}_{22}(z) - (\hat{H}_{21}(z) - D_{21})(I + M_1\hat{H}_{11}(z))^{-1}M_1(\hat{H}_{12}(z) - D_{12}) - D_{21}(I + M_1\hat{H}_{11}(z))^{-1}M_1(\hat{H}_{12}(z) - D_{12}) \\
&\quad - (\hat{H}_{21}(z) - D_{21})(I + M_1\hat{H}_{11}(z))^{-1}M_1D_{12} - D_{21}(I + M_1\hat{H}_{11}(z))^{-1}M_1D_{12} \\
&= \hat{H}_{22}(z) - \hat{H}_{21}(z)(I + M_1\hat{H}_{11}(z))^{-1}M_1\hat{H}_{12}(z).
\end{aligned}$$

Thus, we get the desired results as (9.5).