Detecting equivalence between iterative algorithms for optimization

Madeleine Udell

Operations Research and Information Engineering
Cornell University

Joint work with
Shipu Zhao (Cornell) and Laurent Lessard (Northeastern)

October 4, 2021

Outline

motivating examples

algorithm representation

oracle equivalence

shift equivalence

algorithm conjugation

*

The importance of novelty

The importance of novelty

An algorithm: Douglas-Rachford splitting

- $x \in \mathbf{R}^n$ variables are algorithm state
- superscript $k \in \mathbf{Z}$ is iteration counter
- $t \in \mathbf{R}$ is stepsize
- function $f : \mathbf{R}^n \to \mathbf{R}$
- an optimization *oracle*: the proximal operator

$$\text{prox}_{tf}(x) = \underset{z}{\text{argmin}}\, tf(z) + \frac{1}{2}\|z - x\|^2 : \mathbf{R}^n \to \mathbf{R}^n$$

Algorithm Douglas-Rachford splitting (1956)

---

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

1. $x_1^{k+1} = \text{prox}_{tf}(x_3^k)$
2. $x_2^{k+1} = \text{prox}_{tg}(2x_1^{k+1} - x_3^k)$
3. $x_3^{k+1} = x_3^k + x_2^{k+1} - x_1^{k+1}$

end for

How to obfuscate an algorithm

- rotate or scale state variable
- add or remove extra state variable
- permute update equations

Algorithm Douglas-Rachford splitting (1956)

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

1. $x_1^{k+1} = \text{prox}_{tf}(x_3^k)$
2. $x_2^{k+1} = \text{prox}_{tg}(2x_1^{k+1} - x_3^k)$
3. $x_3^{k+1} = x_3^k + x_2^{k+1} - x_1^{k+1}$

end for

Algorithm Linear transformation of DR

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

1. $x_1^{k+1} = 2\text{prox}_{tf}(x_3^k)$
2. $x_2^{k+1} = 2\text{prox}_{tg}(x_1^{k+1} - x_3^k)$
3. $x_3^{k+1} = x_3^k + \frac{1}{2}(x_2^{k+1} - x_1^{k+1})$

end for

Algorithm DR removing extra state

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

1. $x_2^{k+1} = \text{prox}_{tg}(2\text{prox}_{tf}(x_3^k) - x_3^k)$
2. $x_3^{k+1} = x_3^k + x_2^{k+1} - \text{prox}_{tf}(x_3^k)$

end for

Algorithm A permutation of DR

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

1. $x_2^{k+1} = \text{prox}_{tg}(2x_1^{k+1} - x_3^k)$
2. $x_3^{k+1} = x_3^k + x_2^{k+1} - x_1^{k+1}$
3. $x_1^{k+1} = \text{prox}_{tf}(x_3^k)$

end for

Novelty? Or not?

- $F : \mathbf{R}^n \to \mathbf{R}^n$ is *oracle* call
  (e.g., gradient of a function or proximal operator)
- $\eta \in \mathbf{R}$ is stepsize

Algorithm Modified Arrow–Hurwicz (1980)

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

1. $x_1^{k+1} = x_1^k - \eta F(x_2^k)$
2. $x_2^{k+1} = x_1^{k+1} - \eta F(x_2^k)$

end for

Algorithm Extrapolation from past (2019)

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

1. $x_2^k = x_1^k - \eta F(x_2^{k-1})$
2. $x_1^{k+1} = x_1^k - \eta F(x_2^k)$

end for

Algorithm Optimistic mirror descent (2013)

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

1. $x_2^{k+1} = x_2^k - 2\eta F(x_2^k) + \eta F(x_2^{k-1})$

end for

Algorithm Reflected gradient method (2015)

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

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

end for

First pair

Algorithm Modified Arrow–Hurwicz (1980)

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

1. $x_1^{k+1} = x_1^k - \eta F(x_2^k)$
2. $x_2^{k+1} = x_1^{k+1} - \eta F(x_2^k)$

end for

Algorithm Extrapolation from past (2019)

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

1. $x_2^k = x_1^k - \eta F(x_2^{k-1})$
2. $x_1^{k+1} = x_1^k - \eta F(x_2^k)$

end for

First pair

Algorithm Modified Arrow–Hurwicz (1980)

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

1. $x_1^{k+1} = x_1^k - \eta F(x_2^k)$
2. $x_2^{k+1} = x_1^{k+1} - \eta F(x_2^k)$

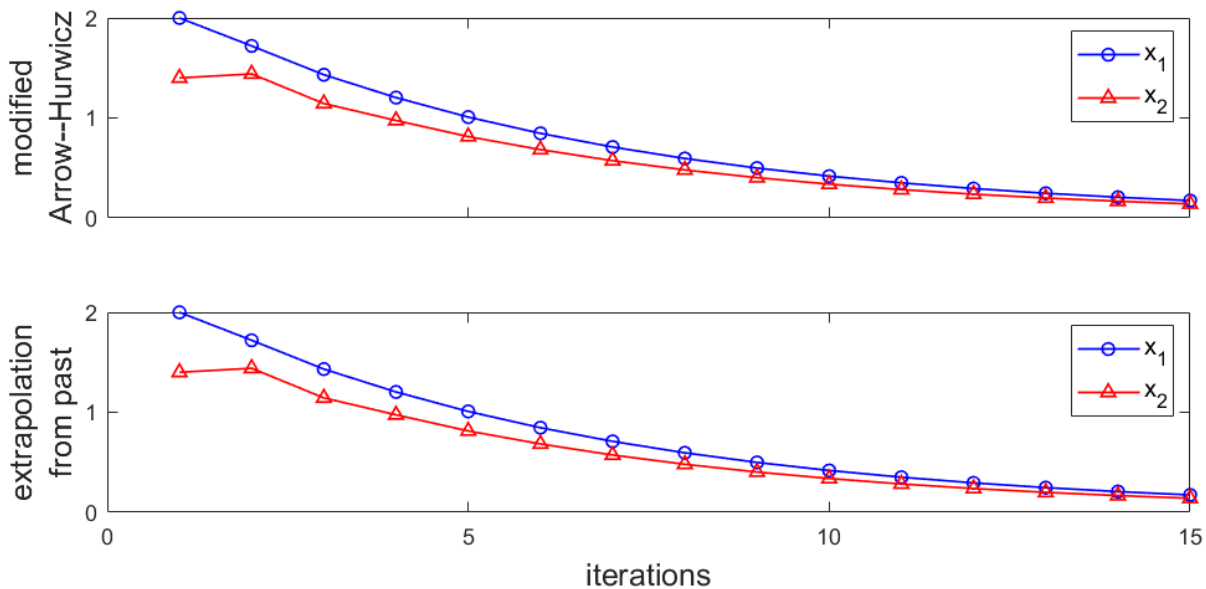
end for

Algorithm Extrapolation from past (2019)

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

1. $x_2^k = x_1^k - \eta F(x_2^{k-1})$
2. $x_1^{k+1} = x_1^k - \eta F(x_2^k)$

end for



- State sequences are identical!

First pair

Algorithm Modified Arrow–Hurwicz (1980)

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

1. $x_1^{k+1} = x_1^k - \eta F(x_2^k)$
2. $x_2^{k+1} = x_1^{k+1} - \eta F(x_2^k)$

end for

Algorithm Extrapolation from past (2019)

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

1. $x_2^k = x_1^k - \eta F(x_2^{k-1})$
2. $x_1^{k+1} = x_1^k - \eta F(x_2^k)$

end for

Definition (state equivalence)

Algorithms are state-equivalent, if algorithms have the same number of states, and the state sequences are equivalent up to an invertible linear transformation.

Second pair

Algorithm Modified Arrow–Hurwicz (1980)

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

1. $x_1^{k+1} = x_1^k - \eta F(x_2^k)$
2. $x_2^{k+1} = x_1^{k+1} - \eta F(x_2^k)$

end for

Algorithm Optimistic mirror descent (2013)

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

1. $x_2^{k+1} = x_2^k - 2\eta F(x_2^k) + \eta F(x_2^{k-1})$

end for

Second pair

Algorithm Modified Arrow–Hurwicz (1980)

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

1. $x_1^{k+1} = x_1^k - \eta F(x_2^k)$
2. $x_2^{k+1} = x_1^{k+1} - \eta F(x_2^k)$

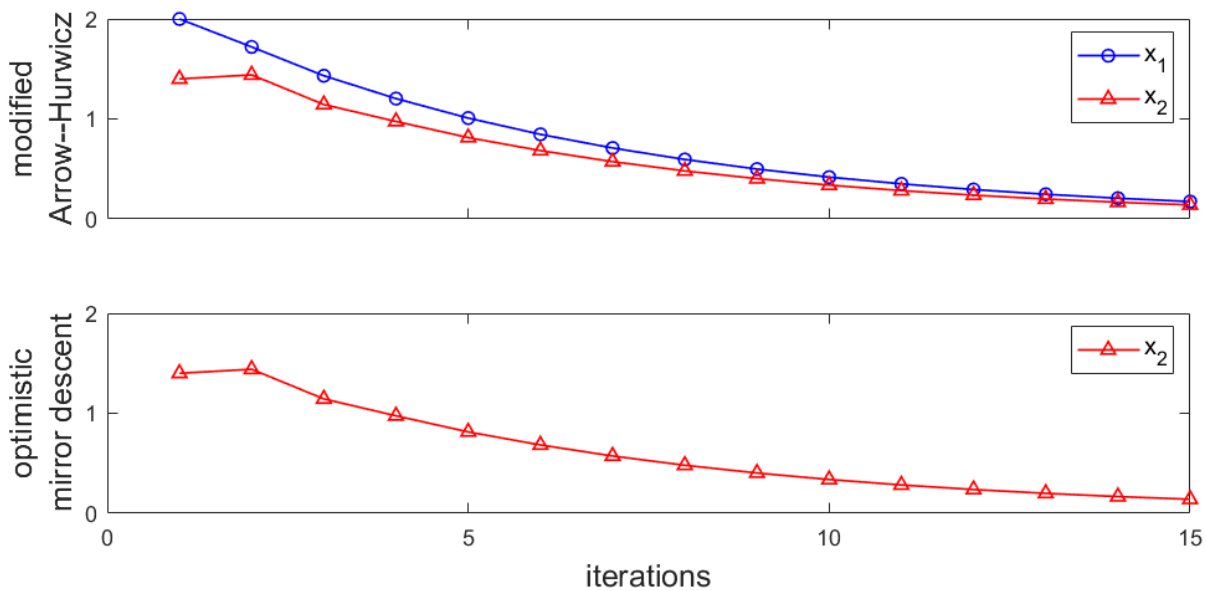
end for

Algorithm Optimistic mirror descent (2013)

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

1. $x_2^{k+1} = x_2^k - 2\eta F(x_2^k) + \eta F(x_2^{k-1})$

end for



- related by removing one state

Second pair

Algorithm Modified Arrow–Hurwicz (1980)

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

1. $x_1^{k+1} = x_1^k - \eta F(x_2^k)$
2. $x_2^{k+1} = x_1^{k+1} - \eta F(x_2^k)$

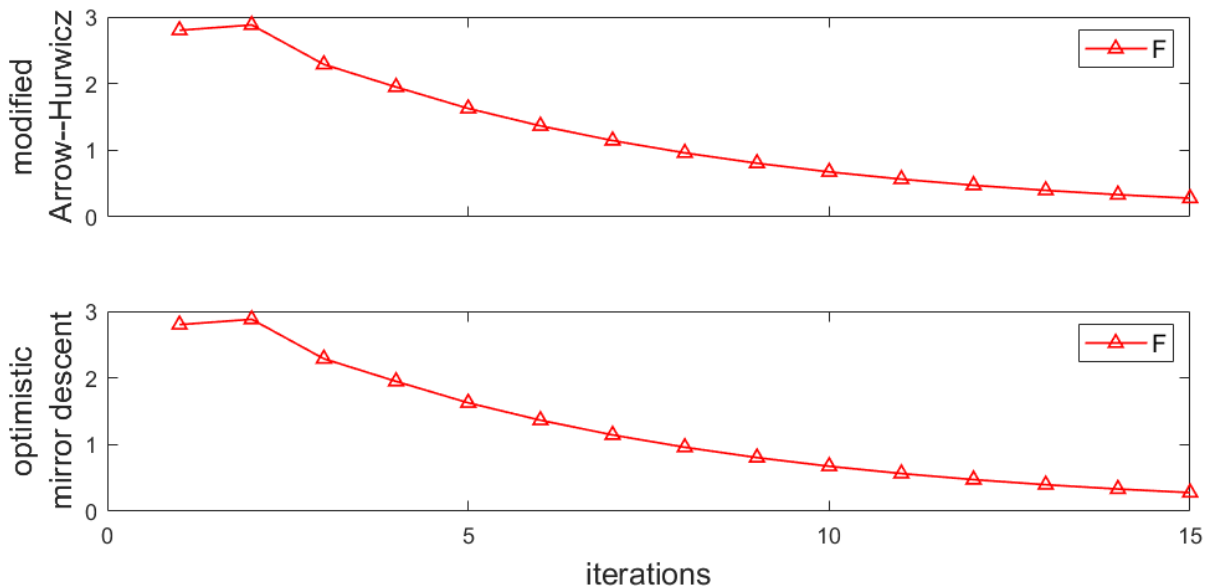
end for

Algorithm Optimistic mirror descent (2013)

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

1. $x_2^{k+1} = x_2^k - 2\eta F(x_2^k) + \eta F(x_2^{k-1})$

end for



- sequence of oracle calls $F(\cdot)$ is identical

Second pair

Algorithm Modified Arrow–Hurwicz (1980)

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

1. $x_1^{k+1} = x_1^k - \eta F(x_2^k)$
2. $x_2^{k+1} = x_1^{k+1} - \eta F(x_2^k)$

end for

Algorithm Optimistic mirror descent (2013)

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

1. $x_2^{k+1} = x_2^k - 2\eta F(x_2^k) + \eta F(x_2^{k-1})$

end for

- Oracle sequences are identical!

Second pair

Algorithm Modified Arrow–Hurwicz (1980)

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

1. $x_1^{k+1} = x_1^k - \eta F(x_2^k)$
2. $x_2^{k+1} = x_1^{k+1} - \eta F(x_2^k)$

end for

Algorithm Optimistic mirror descent (2013)

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

1. $x_2^{k+1} = x_2^k - 2\eta F(x_2^k) + \eta F(x_2^{k-1})$

end for

- Oracle sequences are identical!

Definition (Oracle equivalence)

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.

A surprising example: DR and ADMM

Algorithm Douglas-Rachford splitting (1956)

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

1. $x_1^{k+1} = \text{prox}_{tf}(x_3^k)$
2. $x_2^{k+1} = \text{prox}_{tg}(2x_1^{k+1} - x_3^k)$
3. $x_3^{k+1} = x_3^k + x_2^{k+1} - x_1^{k+1}$

end for

Algorithm Alternating direction method of multipliers

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

1. $\xi_1^{k+1} = \text{argmin}_\xi \{g(\xi) + \frac{\rho}{2} \left\| A\xi + B\xi_2^k - c + \xi_3^k \right\|^2\}$
2. $\xi_2^{k+1} = \text{argmin}_\xi \{f(\xi) + \frac{\rho}{2} \left\| A\xi_1^{k+1} + B\xi - c + \xi_3^k \right\|^2\}$
3. $\xi_3^{k+1} = \xi_3^k + A\xi_1^{k+1} + B\xi_2^{k+1} - c$

end for

A surprising example: DR and ADMM

Algorithm Douglas-Rachford splitting (1956)

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

1. $x_1^{k+1} = \text{prox}_{tf}(x_3^k)$
2. $x_2^{k+1} = \text{prox}_{tg}(2x_1^{k+1} - x_3^k)$
3. $x_3^{k+1} = x_3^k + x_2^{k+1} - x_1^{k+1}$

end for

Algorithm Alternating direction method of multipliers

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

1. $\xi_1^{k+1} = \text{argmin}_\xi \{g(\xi) + \frac{\rho}{2} \left\| A\xi + B\xi_2^k - c + \xi_3^k \right\|^2\}$
2. $\xi_2^{k+1} = \text{argmin}_\xi \{f(\xi) + \frac{\rho}{2} \left\| A\xi_1^{k+1} + B\xi - c + \xi_3^k \right\|^2\}$
3. $\xi_3^{k+1} = \xi_3^k + A\xi_1^{k+1} + B\xi_2^{k+1} - c$

end for

- Suppose $A = I$, $B = -I$, $c = 0$, and $\rho = 1/t$ in ADMM.

A surprising example: DR and ADMM

Algorithm Douglas-Rachford splitting (1956)

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

1. $x_1^{k+1} = \text{prox}_{tf}(x_3^k)$
2. $x_2^{k+1} = \text{prox}_{tg}(2x_1^{k+1} - x_3^k)$
3. $x_3^{k+1} = x_3^k + x_2^{k+1} - x_1^{k+1}$

end for

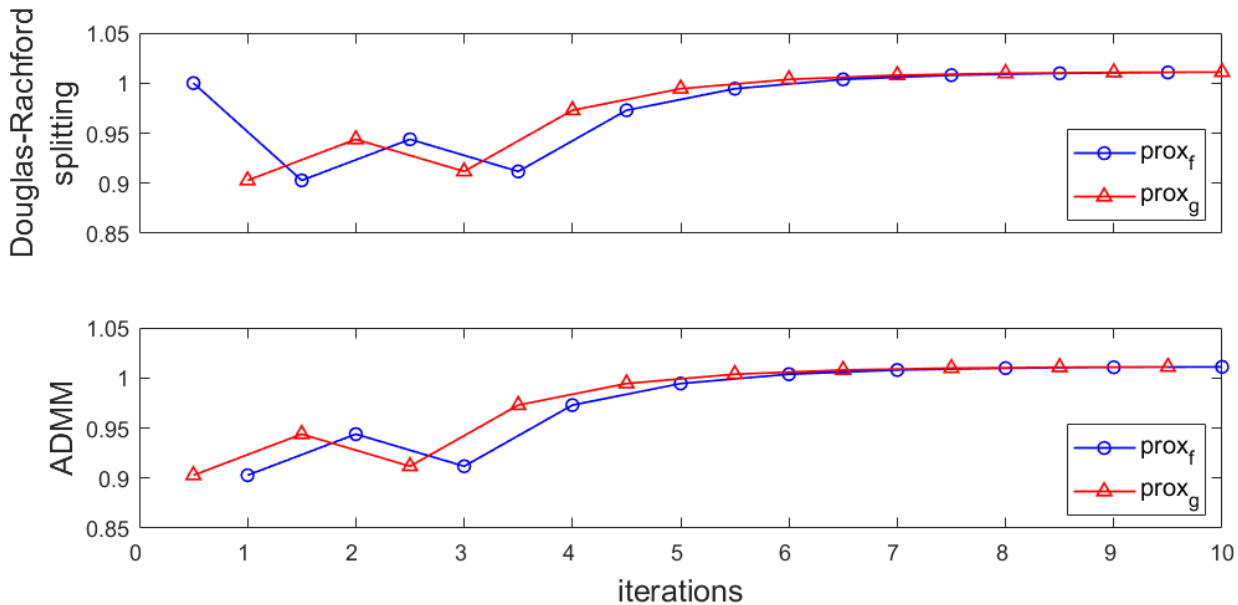
Algorithm Alternating direction method of multipliers

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

1. $\xi_1^{k+1} = \text{argmin}_\xi \{g(\xi) + \frac{\rho}{2} \left\| A\xi + B\xi_2^k - c + \xi_3^k \right\|^2\}$
2. $\xi_2^{k+1} = \text{argmin}_\xi \{f(\xi) + \frac{\rho}{2} \left\| A\xi_1^{k+1} + B\xi - c + \xi_3^k \right\|^2\}$
3. $\xi_3^{k+1} = \xi_3^k + A\xi_1^{k+1} + B\xi_2^{k+1} - c$

end for

- Suppose $A = I$, $B = -I$, $c = 0$, and $\rho = 1/t$ in ADMM.



Douglas-Rachford splitting and ADMM

Algorithm Douglas-Rachford splitting (1956)

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

1. $x_1^{k+1} = \text{prox}_{tf}(x_3^k)$
2. $x_2^{k+1} = \text{prox}_{tg}(2x_1^{k+1} - x_3^k)$
3. $x_3^{k+1} = x_3^k + x_2^{k+1} - x_1^{k+1}$

end for

Algorithm Alternating direction method of multipliers

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

1. $\xi_1^{k+1} = \text{argmin}_\xi \{g(\xi) + \frac{\rho}{2} \left\| A\xi + B\xi_2^k - c + \xi_3^k \right\|^2\}$
2. $\xi_2^{k+1} = \text{argmin}_\xi \{f(\xi) + \frac{\rho}{2} \left\| A\xi_1^{k+1} + B\xi - c + \xi_3^k \right\|^2\}$
3. $\xi_3^{k+1} = \xi_3^k + A\xi_1^{k+1} + B\xi_2^{k+1} - c$

end for

- Not state-equivalent or oracle-equivalent.
- Oracle sequences match after shifting ADMM one step backward.

Douglas-Rachford splitting and ADMM

Algorithm Douglas-Rachford splitting (1956)

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

1. $x_1^{k+1} = \text{prox}_{tf}(x_3^k)$
2. $x_2^{k+1} = \text{prox}_{tg}(2x_1^{k+1} - x_3^k)$
3. $x_3^{k+1} = x_3^k + x_2^{k+1} - x_1^{k+1}$

end for

Algorithm Alternating direction method of multipliers

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

1. $\xi_1^{k+1} = \text{argmin}_\xi \{g(\xi) + \frac{\rho}{2} \left\| A\xi + B\xi_2^k - c + \xi_3^k \right\|^2\}$
2. $\xi_2^{k+1} = \text{argmin}_\xi \{f(\xi) + \frac{\rho}{2} \left\| A\xi_1^{k+1} + B\xi - c + \xi_3^k \right\|^2\}$
3. $\xi_3^{k+1} = \xi_3^k + A\xi_1^{k+1} + B\xi_2^{k+1} - c$

end for

- Not state-equivalent or oracle-equivalent.
- Oracle sequences match after shifting ADMM one step backward.

Definition (Shift equivalence)

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.

Related work

search over mathematical objects

- The On-Line Encyclopedia of Integer Sequences
- using deep learning (e.g. GPT3) to generate code snippets
- ...

representing optimization in standard form for computer analysis

- CVX Grant & Boyd 2014
- analysis of first order methods Lessard, Recht & Packard 2016
- ...

unified views of operator splitting methods

- Ryu & Boyd 2016, Ryu & Yin 2020
- ...

Outline

motivating examples

algorithm representation

oracle equivalence

shift equivalence

algorithm conjugation

*

Algorithm as linear system with feedback

- x^k state
- u^k input to linear system (output from oracle)
- y^k output to linear system (input to oracle)
- nonlinear feedback ϕ represents the set of oracles

$$
\begin{aligned}
x^{k+1} &= Ax^k + Bu^k && \triangleright \text{state update} \\
y^k &= Cx^k + Du^k && \triangleright \text{output} \\
u^k &= \phi(y^k) && \triangleright \text{oracle}
\end{aligned}
$$

Algorithm as linear system with feedback

- x^k state
- u^k input to linear system (output from oracle)
- y^k output to linear system (input to oracle)
- nonlinear feedback ϕ represents the set of oracles

$$
\begin{aligned}
x^{k+1} &= Ax^k + Bu^k && \triangleright \text{state update} \\
y^k &= Cx^k + Du^k && \triangleright \text{output} \\
u^k &= \phi(y^k) && \triangleright \text{oracle}
\end{aligned}
$$

implementation order

- if $D = 0$, compute oracle, then state update, then output
- if $D \neq 0$, oracle requires joint solve of output and oracle equation
  (e.g., prox or argmin)

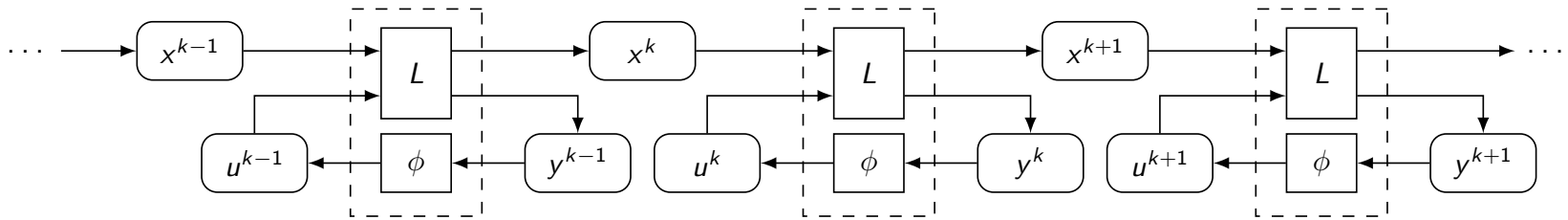
Algorithm as linear system with feedback

- x^k state
- u^k input to linear system (output from oracle)
- y^k output to linear system (input to oracle)
- nonlinear feedback ϕ represents the set of oracles

$$
\begin{aligned}
x^{k+1} &= Ax^k + Bu^k && \triangleright \text{state update} \\
y^k &= Cx^k + Du^k && \triangleright \text{output} \\
u^k &= \phi(y^k) && \triangleright \text{oracle}
\end{aligned}
$$

- define *state-space realization* L (first two equations)

block-diagram representation:



Example: modified Arrow–Hurwicz

```python
import linnaeus as lin
from linnaeus import Algorithm

ah = Algorithm("modified Arrow--Hurwicz")
x1, x2 = ah.add_var("x1", "x2")
F = ah.add_oracle("F")
eta = ah.add_parameter("eta")

ah.add_update(x1, x1 - eta * F(x2))
ah.add_update(x2, x1 - eta * F(x2))

ah.parse()
ah.get_ss(verbose = True)
```

```
------------------------------------------------------------
State-space realization:
```

$$\begin{bmatrix} x_1^+ \\ x_2^+ \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} -\eta \\ -2\eta \end{bmatrix} [F(y_0)]$$

$$[y_0] = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + [0] [F(y_0)]$$

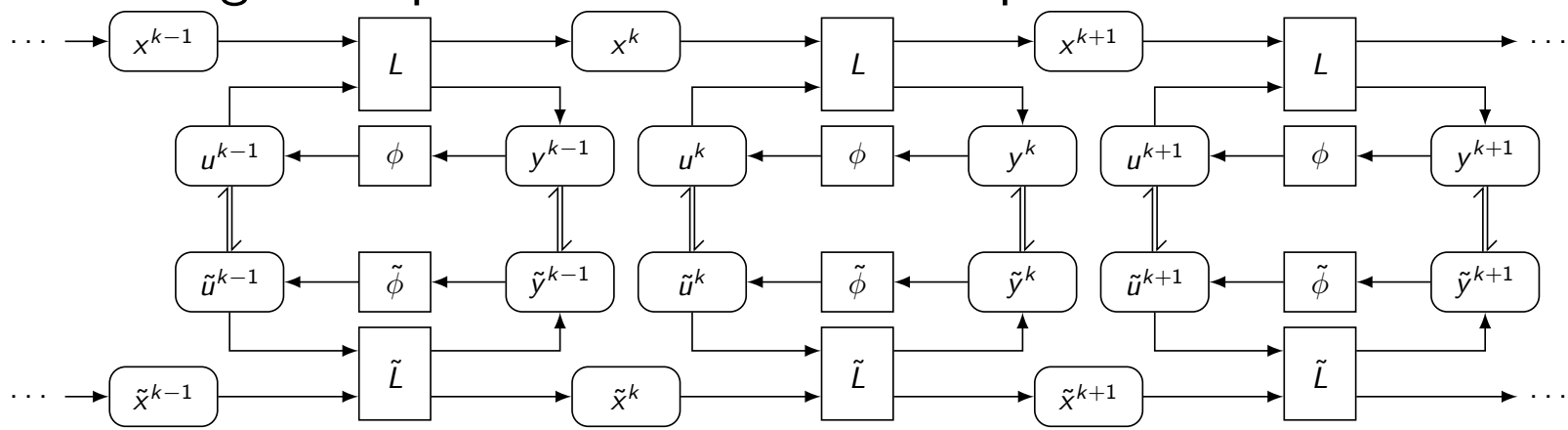
```
------------------------------------------------------------
```

Oracle equivalence

consider:

- algorithm $\mathcal{A}$ with state-space realization L and oracle ϕ
- algorithm $\tilde{\mathcal{A}}$ with state-space realization $\tilde{L}$ and oracle $\tilde{\phi}$

block-diagram representation of oracle equivalence:



If inputs match ($u = \tilde{u}$) and outputs match ($y = \tilde{y}$), then algorithms $\mathcal{A}$ and $\tilde{\mathcal{A}}$ are indistinguishable from the point of view of the oracle.

Why oracle equivalence?

Analytical properties involving oracle sequences are preserved!

Why oracle equivalence?

Analytical properties involving oracle sequences are preserved!

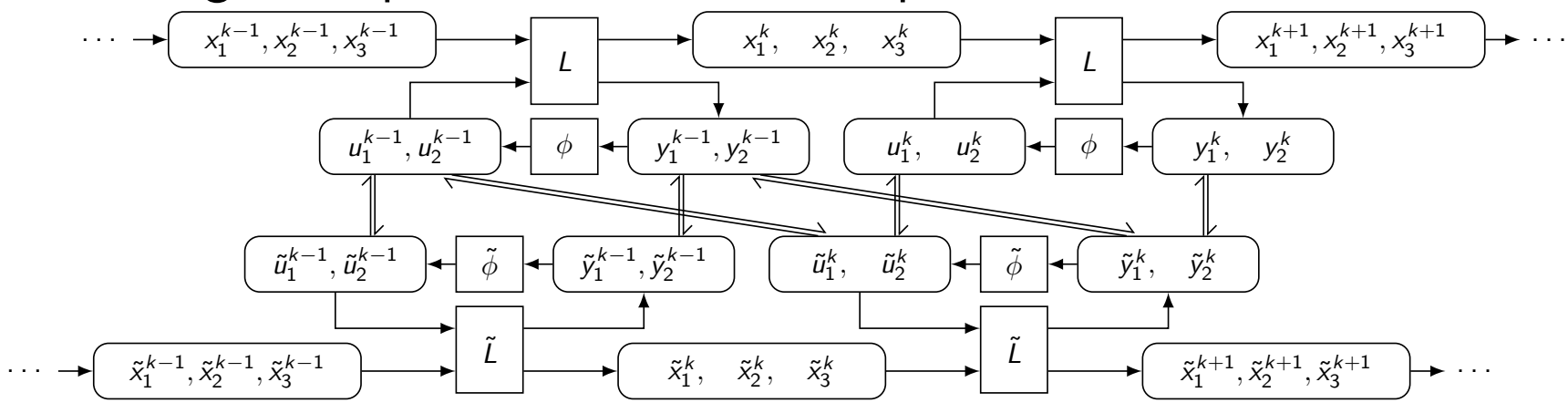
- convergence, e.g. convergence rate, fixed point
- robustness, e.g. worst-case convergence
- noisy oracle, e.g. additive or multiplicative noise, or even adversarial noise

Shift equivalence

consider

- algorithm $\mathcal{A}$ with state-space realization L and oracle ϕ
- algorithm $\tilde{\mathcal{A}}$ with state-space realization $\tilde{L}$ and oracle $\tilde{\phi}$

block-diagram representation of shift equivalence:

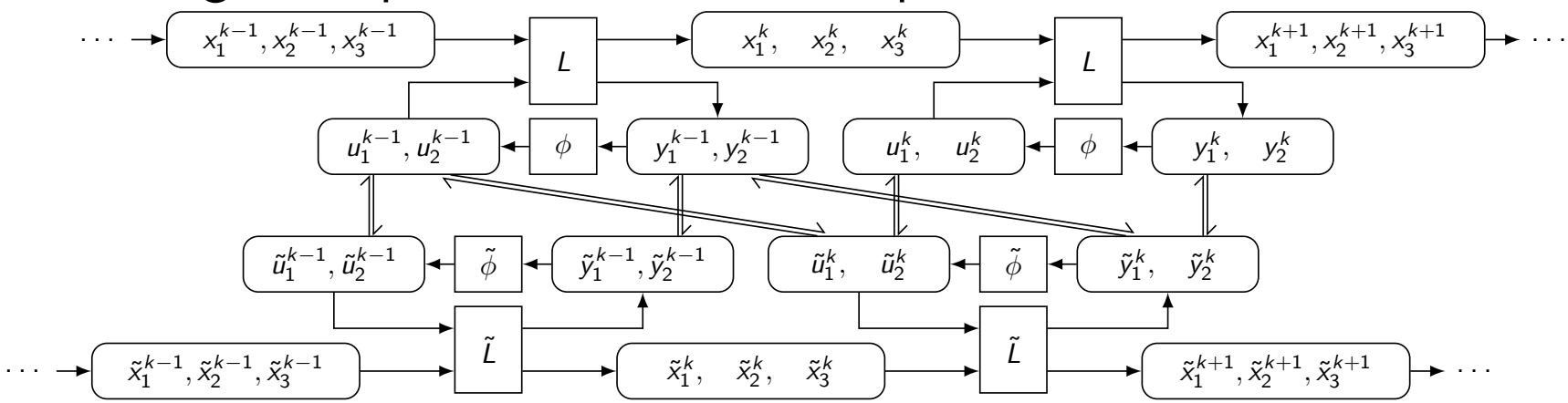


Shift equivalence

consider

- algorithm $\mathcal{A}$ with state-space realization L and oracle ϕ
- algorithm $\tilde{\mathcal{A}}$ with state-space realization $\tilde{L}$ and oracle $\tilde{\phi}$

block-diagram representation of shift equivalence:



- Shift equivalence generalizes oracle equivalence.
- Analytical properties involving oracle sequences are also preserved.

Outline

motivating examples

algorithm representation

oracle equivalence

shift equivalence

algorithm conjugation

*

How to identify equivalence?

How to identify equivalence?

check entire input and output sequences for every initialization?

How to identify equivalence?

check entire input and output sequences for every initialization?

- usually impossible

How to identify equivalence?

check entire input and output sequences for every initialization?

- usually impossible

our approach:

1. parse each algorithm to standard form
   (as linear system in feedback with nonlinearity)
2. compute transfer function of each linear system
3. use computer algebra system to solve for parameter values
   (if any) that make the transfer functions equal

Transfer functions

input-output map:

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

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

focus on zero state response. define

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

so

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

Transfer functions

introduce variable $z \in \mathbf{R}$ and take z-transform:

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

define the *transfer function* $\hat{H}(z)$

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

Transfer functions

introduce variable $z \in \mathbf{R}$ and take z-transform:

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

define the *transfer function* $\hat{H}(z)$

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

the transfer function $\hat{H}(z)$

- uniquely characterizes the input-output map of a system
- is fast to compute

Transfer functions are invariant under linear transformations

- linear system L with state-space realization (A, B, C, D)
- invertible linear transformation $Q : \mathbf{R}^n \to \mathbf{R}^n$

map $x \to Qx$ and write state space realization:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \to \begin{bmatrix} QAQ^{-1} & QB \\ CQ^{-1} & D \end{bmatrix}$$

how does transfer function change? it's the same!

$$\hat{H}'(z) = CQ^{-1}(zI - QAQ^{-1})^{-1}QB + D = C(zI - A)^{-1}B + D = \hat{H}(z)$$

- Transfer functions are invariant under invertible linear transformations of state variables.

Example: modified Arrow–Hurwicz

```
ah.get_tf(verbose = True)
```

```
------------------------------------------------------------
Transfer function:
```

$$\left[\frac{\eta(1-2z)}{z(z-1)}\right]$$

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

map $x \to Qx$ with $Q = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$:

```
ah2 = Algorithm("linear transform of modified Arrow--Hurwicz")
x1, x2 = ah2.add_var("x1", "x2")
F = ah2.add_oracle("F")
eta = ah2.add_parameter("eta")
ah2.add_update(x1, 2 * x1 - 2 * x2 - 3 * eta * F(x2))
ah2.add_update(x2,  1/2 * x1 - 1/2 * eta * F(x2))
ah2.parse()
ah2.get_tf(verbose = True)
```

```
------------------------------------------------------------
Transfer function:
```

$$\left[\frac{\eta(1-2z)}{z(z-1)}\right]$$

Transfer function characterizes oracle equivalence

Proposition

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

Transfer function characterizes oracle equivalence

Proposition

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

- proof: transfer function uniquely characterizes input-output map

Transfer function characterizes oracle equivalence

Proposition

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

- proof: transfer function uniquely characterizes input-output map
- note oracle-equivalent algorithms can have a different number of state variables.

Modified Arrow–Hurwicz and extrapolation from the past

```
ep = Algorithm("extrapolation from the past")
x1, x2, F_prev = ep.add_var("x1", "x2", "F_prev")
F = ep.add_oracle("F")
eta = ep.add_parameter("eta")
ep.add_update(x2, x1 - eta * F_prev)
ep.add_update(x1, x1 - eta * F(x2))
ep.add_update(F_prev, F(x2))
ep.parse()

lin.is_equivalent(ah, ep, verbose = True)
```

```
------------------------------------------------------------
Parse extrapolation from the past:
```

$$x_2 \leftarrow x_1 - \eta F_{prev}$$
$$x_1 \leftarrow x_1 - \eta F(x_2)$$
$$F_{prev} \leftarrow F(x_2)$$

```
------------------------------------------------------------
------------------------------------------------------------
modified Arrow--Hurwicz is equivalent to extrapolation from the past.
------------------------------------------------------------
True
```

Modified Arrow–Hurwicz and optimistic mirror descent

```
md = Algorithm("optimistic mirror descent")
x2, F_prev = md.add_var("x2", "F_prev")
F = md.add_oracle("F")
eta = md.add_parameter("eta")
md.set_auto(False)
md.add_update(x2, x2 - 2 * eta * F(x2) + eta * F_prev)
md.add_update(F_prev, F(x2))
md.parse()

lin.is_equivalent(ah, md, verbose = True)
```

```
------------------------------------------------------------
Parse optimistic mirror descent:
```

$$x_2^+ = x_2 - 2\eta F(x_2) + \eta F_{prev}$$
$$F_{prev}^+ = F(x_2)$$

```
------------------------------------------------------------
------------------------------------------------------------
modified Arrow--Hurwicz is equivalent to optimistic mirror descent.
------------------------------------------------------------
True
```

Outline

motivating examples

algorithm representation

oracle equivalence

shift equivalence

algorithm conjugation

*

How to check shift equivalence?

Check entire input and output sequences for every initialization?

How to check shift equivalence?

Check entire input and output sequences for every initialization?

- Impossible.

How to check shift equivalence?

Check entire input and output sequences for every initialization?

- Impossible.

Check how transfer functions changes under shifts!

Cyclic permutation of update equations

Proposition (Cyclic permutation and shift equivalence)

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.)

Example: Douglas-Rachford splitting

```
DR = Algorithm("Douglas-Rachford splitting")
x1, x2, x3 = DR.add_var("x1", "x2", "x3")
t = DR.add_parameter("t")
f, g = DR.add_function("f", "g")
DR.add_update(x1, lin.prox(f, t)(x3))
DR.add_update(x2, lin.prox(g, t)(2 * x1 - x3))
DR.add_update(x3, x3 + x2 - x1)
DR.parse()
```

```
DR1 = Algorithm("First permutation of Douglas-Rachford splitting")
x1, x2, x3 = DR1.add_var("x1", "x2", "x3")
t = DR1.add_parameter("t")
f, g = DR1.add_function("f", "g")
DR1.add_update(x2, lin.prox(g, t)(2 * x1 - x3))
DR1.add_update(x3, x3 + x2 - x1)
DR1.add_update(x1, lin.prox(f, t)(x3))
DR1.parse()

DR2 = Algorithm("Second permutation of Douglas-Rachford splitting")
x1, x2, x3 = DR2.add_var("x1", "x2", "x3")
t = DR2.add_parameter("t")
f, g = DR2.add_function("f", "g")
DR2.add_update(x3, x3 + x2 - x1)
DR2.add_update(x1, lin.prox(f, t)(x3))
DR2.add_update(x2, lin.prox(g, t)(2 * x1 - x3))
DR2.parse()
```

Example: Douglas-Rachford splitting

```
lin.is_permutation(DR, DR1)
```

```
True
```

Example: Douglas-Rachford splitting

```
lin.is_permutation(DR, DR1)
```

```
True
```

```
lin.is_equivalent(DR, DR1)
```

```
False
```

Example: Douglas-Rachford splitting

```
lin.is_permutation(DR, DR1)
```

```
True
```

```
lin.is_equivalent(DR, DR1)
```

```
False
```

```
lin.is_equivalent(DR, DR2)
```

```
True
```

Notation: cyclic permutations

- algorithm $\mathcal{A}$
- sequence of oracle indices (n)
- cyclic permutation $\pi = (j+1, \ldots, n, 1, \ldots, j)$

Partition the state-space realization $L_{\mathcal{A}}$ and transfer function $\hat{H}_{\mathcal{A}}(z)$ with respect to $(1, \ldots, j)$ and $(j+1, \ldots, n)$.

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

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

Transfer function characterizes shift equivalence

Proposition

Assume $D_{12} = 0$. Then algorithm $\mathcal{B}$ is a cyclic permutation of algorithm $\mathcal{A}$ with respect to π if and only if the transfer function of $\mathcal{B}$ satisfies

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

Transfer function characterizes shift equivalence

Proposition

Assume $D_{12} = 0$. Then algorithm $\mathcal{B}$ is a cyclic permutation of algorithm $\mathcal{A}$ with respect to π if and only if the transfer function of $\mathcal{B}$ satisfies

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

- D is lower-triangular if and only if algorithm is causal.
- So $D_{12} = 0$ is a weak assumption: all implementable algorithms are causal.

Douglas-Rachford splitting and ADMM

```
ADMM = Algorithm("ADMM")
f, g = ADMM.add_function("f", "g")
rho = ADMM.add_parameter("rho")
A, B, c = ADMM.add_parameter("A", "B", "c", commutative = False)
x1, x2, x3 = ADMM.add_var("x1", "x2", "x3")
ADMM.add_update(x1, lin.argmin(x1, f(x1) + 1/2 * rho * lin.
 ↪norm_square(A * x1 + B * x2 + x3 - c)))
ADMM.add_update(x2, lin.argmin(x2, g(x2) + 1/2 * rho * lin.
 ↪norm_square(A * x1 + B * x2 + x3 - c)))
ADMM.add_update(x3, x3 + A * x1 + B * x2 - c)
ADMM.parse()

lin.test_permutation(DR, ADMM)
```

```
------------------------------------------------------------
============================================================
Parameters of Douglas-Rachford splitting:
```

$$t$$

```
Parameters of ADMM:
```

$$\rho, A, B, c$$

```
Douglas-Rachford splitting is a permutation of ADMM,
if the parameters satisfy:
```

$$\rho = \rho, \quad A = -\sqrt{\frac{1}{\rho t}}, \quad B = \sqrt{\frac{1}{\rho t}}, \quad c = c_n$$

Outline

motivating examples

algorithm representation

oracle equivalence

shift equivalence

algorithm conjugation

*

Can we obfuscate the oracles?

- function $f : \mathbf{R}^n \to \mathbf{R}^n$

Can we obfuscate the oracles?

- function $f : \mathbf{R}^n \to \mathbf{R}^n$
- Fenchel conjugate $f^*(y) = \sup_x \{x^T y - f(x)\}$

Can we obfuscate the oracles?

- function $f : \mathbf{R}^n \to \mathbf{R}^n$
- Fenchel conjugate $f^*(y) = \sup_x \{x^T y - f(x)\}$
- can compute oracle involving f by using f^* instead:

$$(\partial f)^{-1} = \partial f^* \qquad \text{and} \qquad I - \mathbf{prox}_f = \mathbf{prox}_{f^*}$$

Can we obfuscate the oracles?

- function $f : \mathbf{R}^n \to \mathbf{R}^n$
- Fenchel conjugate $f^*(y) = \sup_x\{x^T y - f(x)\}$
- can compute oracle involving f by using f^* instead:

$$(\partial f)^{-1} = \partial f^* \qquad \text{and} \qquad I - \mathbf{prox}_f = \mathbf{prox}_{f^*}$$

algorithm conjugation: rewrite algorithm using conjugate oracles

Example: conjugate proximal gradient

- rewrite using *Moreau's identity*, $I - \mathbf{prox}_f = \mathbf{prox}_{f^*}$.

Algorithm Proximal gradient

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

1. $x^{k+1} = \text{prox}_{tg}(x^k - t\nabla f(x^k))$

end for

Algorithm Conjugate to proximal gradient

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

1. $\xi^{k+1} = \xi^k - t\nabla f(\xi^k) - t\text{prox}_{\frac{1}{t}g^*}(\frac{1}{t}(\xi^k - t\nabla f(\xi^k)))$

end for

How to characterize conjugation?

first simplify:

- rewrite all oracles in terms of (sub)gradients

How to characterize conjugation?

first simplify:

- rewrite all oracles in terms of (sub)gradients
- e.g. for prox, $u = \mathbf{prox}_f(y) \quad \iff \quad y \in u + \partial f(u)$.

now suppose all oracles are subgradients

- recall $(\partial f)^{-1} = \partial f^*$
- so algorithm conjugation swaps the input and output of the linear system

Notation: conjugation

- algorithm $\mathcal{A}$
- set of oracle indices $[n]$
- subset $\kappa \subseteq [n]$ to conjugate

Notation: conjugation

- algorithm $\mathcal{A}$
- set of oracle indices $[n]$
- subset $\kappa \subseteq [n]$ to conjugate
- suppose $\kappa = \{1, \ldots, k\}$ for some $k < n$
  (if not, permute first)

Notation: conjugation

- algorithm $\mathcal{A}$
- set of oracle indices $[n]$
- subset $\kappa \subseteq [n]$ to conjugate
- suppose $\kappa = \{1, \ldots, k\}$ for some $k < n$
  (if not, permute first)

partition the state-space realization and transfer function to separate oracles in κ:

$$\left[\begin{array}{c|cc} A & B_1 & B_2 \\ \hline C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \end{array}\right], \quad \begin{bmatrix} \hat{H}_{11}(z) & \hat{H}_{12}(z) \\ \hat{H}_{21}(z) & \hat{H}_{22}(z) \end{bmatrix}$$

Transfer function characterizes conjugation

Proposition

Suppose D_{11} is invertible. Then algorithm $\mathcal{B}$ is conjugate to $\mathcal{A}$ with respect to oracles in κ if and only if the transfer function of $\mathcal{B}$ is

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

Transfer function characterizes conjugation

Proposition

Suppose D_{11} is invertible. Then algorithm $\mathcal{B}$ is conjugate to $\mathcal{A}$ with respect to oracles in κ if and only if the transfer function of $\mathcal{B}$ is

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

- in general, need a permutation matrix to swap rows and columns so indices in κ come first

Transfer function characterizes conjugation

Proposition

Suppose D_{11} is invertible. Then algorithm $\mathcal{B}$ is conjugate to $\mathcal{A}$ with respect to oracles in κ if and only if the transfer function of $\mathcal{B}$ is

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

- in general, need a permutation matrix to swap rows and columns so indices in κ come first
- D_{11} is invertible $\rightarrow$ the conjugate algorithm is causal.

Douglas-Rachford splitting and Chambolle-Pock

```
CP = Algorithm("Chambolle-Pock method")
f, g = CP.add_function("f", "g")
x1, x2, x3 = CP.add_var("x1", "x2", "x3")
tau, sigma = CP.add_parameter("tau", "sigma")
CP.add_update(x3, x1)
CP.add_update(x1, lin.prox(f, tau)(x1 - tau*x2))
CP.add_update(x2, lin.prox(g, sigma)(x2 + sigma*(2*x1 - x3)))
CP.parse()

lin.test_conjugation(DR, CP)
```

```
------------------------------------------------------------
============================================================
Parameters of Douglas-Rachford splitting:
```

$$t$$

```
Parameters of Chambolle-Pock method:
```

$$\tau, \sigma$$

```
Douglas-Rachford splitting is conjugate to Chambolle-Pock method,
if the parameters satisfy:
```

$$\tau = t$$

$$\sigma = \frac{1}{t}$$

```
============================================================
------------------------------------------------------------
```

Conjugation and permutation

Proposition (Commutativity)

Conjugation of different oracles commutes. Further, conjugation and cyclic permutation commute.

- important to detect relations between algorithms efficiently.

Douglas-Rachford splitting and ADMM

```
lin.test_conjugate_permutation(DR, ADMM)
```

```
------------------------------------------------------------
============================================================
Parameters of Douglas-Rachford splitting:
```

$$t$$

```
Parameters of ADMM:
```

$$\rho, A, B, c$$

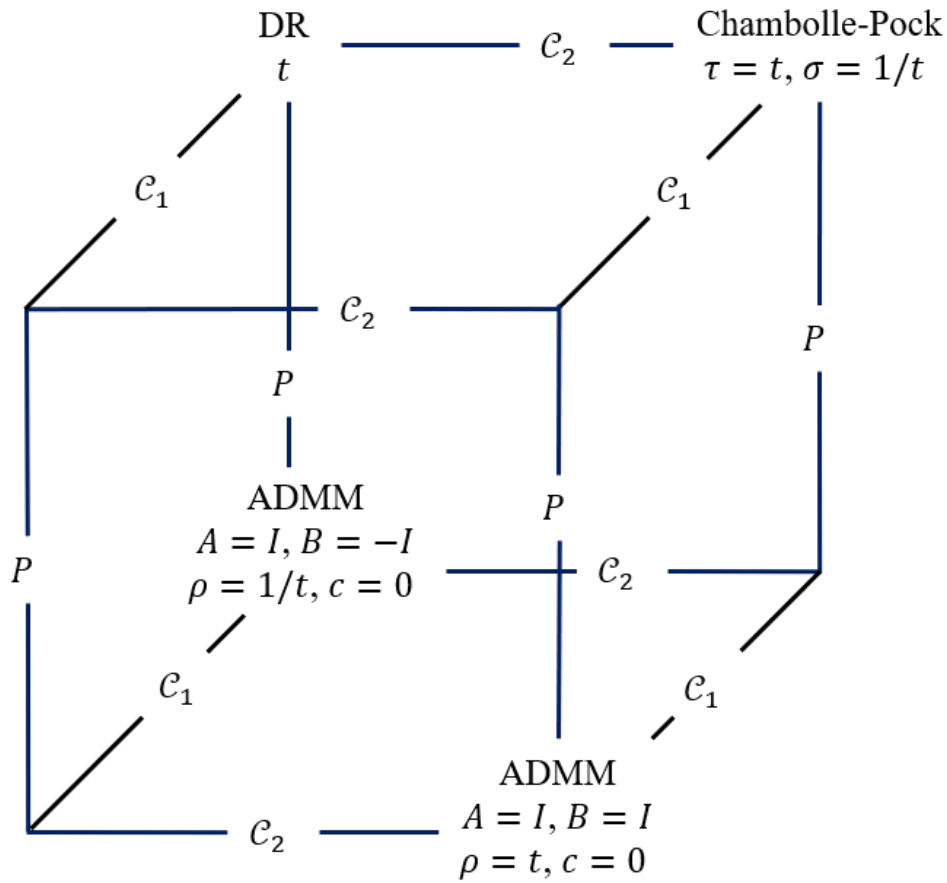
```
Douglas-Rachford splitting is a conjugate permutation of ADMM,
if the parameters satisfy:
```

$$\rho = \rho, \quad A = -\sqrt{\frac{t}{\rho}}$$

$$B = -\sqrt{\frac{t}{\rho}}, \quad c = c_n$$

```
============================================================
------------------------------------------------------------
```

Relations between DR, ADMM, and Chambolle-Pock



Implementation: how does it work?

LINNAEUS `https://github.com/udellgroup/Linnaeus`

- given new algorithm, define variables, oracles, parameters, and update equations
- parse algorithm into standard form (transfer function)
- LINNAEUS includes library of existing algorithms
- use computer algebra system `sympy` to solve for parameter values (if any) that make the transfer function match any existing algorithm

Implementation: how does it work?

LINNAEUS `https://github.com/udellgroup/Linnaeus`

- given new algorithm, define variables, oracles, parameters, and update equations
- parse algorithm into standard form (transfer function)
- LINNAEUS includes library of existing algorithms
- use computer algebra system `sympy` to solve for parameter values (if any) that make the transfer function match any existing algorithm

add your own algorithms!

Summary

- a framework for reasoning about algorithm equivalence
- new definitions of equivalence:
  state equivalence, oracle equivalence, shift equivalence, conjugation, etc.
- tractable way to identify algorithm obfuscation:
  linear transformations, permutation, conjugation, etc.
- implementation: LINNAEUS

Future work + references

Can we detect equivalence between

- stochastic or randomized algorithms?
- parallel or distributed algorithms?
- adaptive or nonlinear algorithms?
- ...

references:

- Zhao, S., Lessard, L., and Udell, M. (2021). An automatic system to detect equivalence between iterative algorithms. arXiv:2105.04684
- `https://github.com/udellgroup/Linnaeus`

Outline

motivating examples

algorithm representation

oracle equivalence

shift equivalence

algorithm conjugation

*

References

Grant, M. & Boyd, S. (2014). CVX: Matlab software for disciplined convex programming, version 2.1.

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

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

Ryu, E. K. & Yin, W. (2020). *Large-scale convex optimization via monotone operators*. Draft.