`linrax`: A JAX Compatible, Simplex Method Linear Program Solver

Brendan Gould, Akash Harapanahalli, and Samuel Coogan[1]

***Abstract*— We present `linrax`, the first simplex based linear program (LP) solver in the JAX ecosystem. In many control algorithms, LPs are often automatically generated and solved online in the control loop. This motivates the design of `linrax`, which is especially suited for compilation into a large JAX-based pipeline as a subroutine. We discuss the challenges of implementing a general purpose LP solver under strict design requirements from JAX. Notably, we can solve general problems which may include dependent constraints—something not possible with existing JAX-compatible LP solvers that use first-order techniques and may fail to converge. We demonstrate `linrax`'s utility through several examples, including a robust control synthesis pipeline for nonlinear vehicles using automatic differentiation through a LP-based reachable set framework.**

I. Introduction

Constrained optimization is an important tool in robotics and control and appears in many forms. For example, techniques like Model-Predictive Control (MPC) [1], Control-Lyapunov functions (CLFs) [2], and Control-Barrier functions (CBFs) [3] choose control inputs *at each point in time* as the solution of an optimization problem. Linear programs (LPs) [4] are a valuable subset of optimization problems that are capable of synthesizing plans or control policies [5] and verifying the safe operation of control systems through reachability analysis [6]. Optimization-based control frameworks have been facilitated by modern advances in computational power, including a rich ecosystem of software optimizers [7], [8]. Many such solvers support additional features such as automatic differentiation [9] and GPU parallelization [10], making it possible to embed optimization problems into gradient-based, scalable control pipelines.

JAX is a high-performance computing platform for Python that provides Just-in-Time Compilation (JIT), GPU Parallelization, and automatic differentiation [11], increasingly being used for control applications [12], [13], [14]. Though there exist many JAX libraries for optimization [15], [16] or gradient descent [17], surprisingly few are designed for mathematical programming problems. Those that are [18], [19] utilize first-order gradient based methods and struggle with solution precision and convergence, especially on problems with degenerate constraints. To address this gap, we introduce a JAX implementation of the *simplex algorithm*, a method for solving LPs exactly (up to floating-point accuracy) that works for problems with degenerate constraints, and demonstrate its utility in control algorithms.

*This work was supported in part by the National Science Foundation under awards #2219755 and #2333488 and by the Air Force Office of Scientific Research under Grant FA9550-23-1-0303.

[1]The authors are affiliated with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA {bgould6, aharapan, sam.coogan}@gatech.edu.

A. Contributions

This work presents `linrax`, an open-source implementation of the simplex algorithm in JAX, available on PyPI and GitHub[1]. Our implementation modifies the classic algorithm to satisfy strict JAX *tracibility* requirements, and therefore enjoys JIT compilation, automatic differentiability, and GPU parallelization. It is especially suited for solving relatively small-sized LPs with high precision, and when embedded as a subroutine in more complex JAX programs (as is common in modern control pipelines). Compared to other JAX optimizers, `linrax` easily handles problems with linearly dependent constraints, something which causes convergence delays or even outright failures in gradient-based methods.

In Section II, we use `linrax` to solve a safe control problem by computing a control "nudge" that minimally perturbs a nominal input trajectory to ensure robust safety, leveraging reachability analysis and automatic differentiation through LPs. Section III pinpoints the incompatibility between JAX and traditional simplex-based approaches, namely the handling of dependent constraints. Our solution, "marking" entries of the tableau between the phase one and phase two problems, is described in Section III-C. Finally, Section IV examines the interface of `linrax`, and compares it to existing LP solvers, both in and outside of the JAX ecosystem. We show that `linrax` is competitive with finely-tuned solvers on relatively small problems, and when embedded in a more complex optimization-based pipeline as intended, `linrax` excels compared to these other solvers.

B. Notation

Let $\overline{\mathbb{R}} = \mathbb{R} \cup \{\pm\infty\}$. For $x, y \in \overline{\mathbb{R}}^n$, we use the elementwise partial order, *i.e.*, $x \leq y$ when $x_i \leq y_i$ for every $i = 1, \dots, n$. For $\underline{x} \leq \overline{x}$, define the interval $[\underline{x}, \overline{x}] = \{x \in \overline{\mathbb{R}}^n : \underline{x} \leq x \leq \overline{x}\}$. $\mathbf{I}_n$ is the $n \times n$ identity matrix, and $0_{m\times n}$ ($1_{m\times n}$) is a $m \times n$ matrix of zeros (ones). We use brackets to index arrays: for a matrix A and $i, j \in \mathbb{Z}^+$, $A[i, j]$ is the component of A in the ith row and jth column, $A[i, :]$ is the entire ith row of A, and $A[\text{end}, \text{end}]$ is the entry in the last row and column of A. Given $x, y \in \mathbb{R}^n$, $x_{i:y} \in \mathbb{R}^n$ is the vector $(x_{i:y})_j = x_j$ for every $j \neq i$ and $(x_{i:y})_i = y_i$.

II. Control Nudging for Robust Safety

LPs are a staple in many safety enforcing control algorithms. Here, we use them to modify a nominal control input into a guaranteed safe input using reachability analysis and automatic differentiation through an LP. This section serves as both motivation for `linrax` and our main case study.

[1]The PyPI package name is `linrax`, and the repository url is `https://github.com/gtfactslab/linrax`.

Given a nonlinear control system of the form

$$\dot{x} = f(x, u, w), \tag{1}$$

where $x \in \mathbb{R}^n$ is the state, $u \in \mathbb{R}^p$ is the control, $w \in \mathbb{R}^q$ is a disturbance, and $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^q$ is a locally Lipschitz vector field, reachability analysis aims to compute the set of states the system can reach for initial conditions $\mathcal{X}_0 \subset \mathbb{R}^n$, a feedforward control mapping $u_{\text{ff}} : \mathbb{R} \to \mathbb{R}^p$, and disturbances $\mathcal{W} \subset \mathbb{R}^p$. Formally, the reachable set of (1) at time t is

$$\mathcal{R}(t, \mathcal{X}_0, u_{\text{ff}}, \mathcal{W}) = \{\phi^f(t, x_0, u_{\text{ff}}, \mathbf{w}),\ \mathbf{w} : \mathbb{R} \to \mathcal{W}\},$$

where ϕ^f is the flow of (1) from initial condition $x_0 \in \mathcal{X}_0$ under inputs $u_{\text{ff}}(t)$ and $\mathbf{w}(t)$ at time t. If the full reachable set is safe, *e.g.*, avoids an obstacle $\mathcal{O} \subseteq \mathbb{R}^n$, then the system is robust to initial condition and/or applied disturbances.

Problem 1 (Feedforward control filtering)**.** We seek to minimally perturb a nominal feedforward input $u_{\text{ff}} : [0, T] \to \mathbb{R}^p$ such that the reachable set avoids $\mathcal{O}$, *i.e.*, find u'_{ff} such that

$$\mathcal{R}(t, \mathcal{X}_0, u'_{\text{ff}}, \mathcal{W}) \cap \mathcal{O} = \emptyset, \quad \forall\, t \in [0, T].$$

Example 1. Consider the dynamics of a kinematic bicycle,

$$\begin{aligned} \dot{p}_x &= v\cos(\phi + \beta(u_2)), \quad \dot{\phi} = \frac{v}{\ell_r}\sin(\beta(u_2)), \\ \dot{p}_y &= v\sin(\phi + \beta(u_2)), \quad \dot{v} = u_1, \end{aligned} \tag{2}$$

where $[p_x\ p_y]^\top \in \mathbb{R}^2$ is the x-y displacement of the center of mass (COM), $v \in \mathbb{R}_{\geq 0}$ is its speed, and $\phi \in [-\pi, \pi)$ is the heading angle. Input u_1 is the applied force, u_2 is the angle of the front wheel, and $\beta(u_2) = \arctan\left(\frac{\ell_f}{\ell_f + \ell_r}\tan(u_2)\right)$ is the slip slide angle where ℓ_f (ℓ_r) is the distance between the COM and the front (rear) wheel. For simplicity, we use $\ell_f = \ell_r = 1$, and Euler integrate with $\Delta t = 5 \cdot 10^{-3}$.

Let $o(x) = 3^2 - (p_x - 4)^2 - (p_y - 4)^2$, $\mathcal{O} = \{x \in \mathbb{R}^4 : o(x) \geq 0\}$ denote a circular obstacle. We first generate a nominal control strategy u_{ff} using nonlinear MPC in `casadi` [1], to stabilize the vehicle to the origin while avoiding the obstacle. This is done using LQR with $A = \frac{\partial f}{\partial x}(\mathring{x}, 0, 0)$, $B = \frac{\partial f}{\partial u}(\mathring{x}, 0, 0)$ linearized about the initial state $\mathring{x} = \begin{bmatrix} 8 & 7 & -\frac{2}{\pi} & 2 \end{bmatrix}$ to obtain a feedback matrix K with $Q = \mathbf{I}_4$, $R = \mathbf{I}_2$. Given the feedforward u_{ff} and the feedback matrix K, we analyze the safety of the closed-loop system

$$\dot{x} = f(x, u_{\text{ff}}(t) + K(x(t) - x_{\text{nom}}(t)), w). \tag{3}$$

The MPC trajectory tightly follows the obstacle, meaning disturbances may cause a collision (Figure 1, left) . Using `linrax` and reachability analysis, we modify u_{ff} so that trajectories of (3) robustly avoid the obstacle.

A. Linear Programs for Reachable Set Overapproximation

In practice, the true reachable set is difficult to obtain and approaches instead verify the safety of an *overapproximation* $\overline{\mathcal{R}}(t, \mathcal{X}_0, \mathcal{W}) \supseteq \mathcal{R}(t, \mathcal{X}_0, \mathcal{W})$. One method uses interval analysis [20] to compute $\overline{\mathcal{R}}$ (Figure 1, left). Later work [21], [6] uses *interval refinement* via LPs to produce a less conservative, polytopic overapproximation. We briefly recall this approach here, but refer to the original works for details.

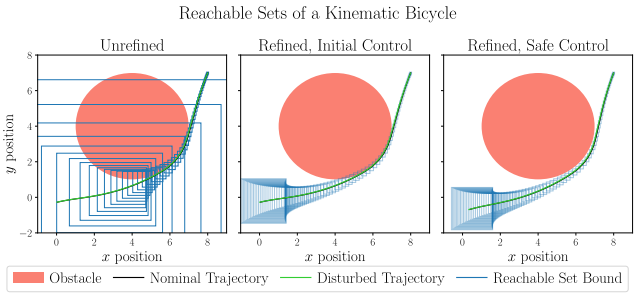

Fig. 1. Overapproximations of the reachable sets of a kinematic bicycle. Using basic interval analysis, the bounds are too conservative for use in safety filtering (**left**). With LP refinement (4), the bounds are tighter, but the feedforward control strategy is unsafe and a sample unsafe trajectory is shown in green (**center**). Algorithm 1 modifies the feedforward control strategy so that the entire reachable set avoids the obstacle (**right**).

Trajectories of (1) sometimes satisfy an invariance condition: $\phi^f(t, x_0, u_{\text{ff}}, \mathbf{w}) \in \mathcal{H}$, arising from some conservation law, artificially introduced by adding model redundancies [21], or by lifting the system to higher dimensions [22]. This property can be combined with any bound $[\underline{y}, \overline{y}]$ from interval analysis to significantly reduce conservatism (especially if $\mathcal{H}$ is well-chosen). Any interval $[\underline{z}, \overline{z}]$ with

$$\mathcal{H} \cap [\underline{y}, \overline{y}] \subseteq [\underline{z}, \overline{z}] \subseteq [\underline{y}, \overline{y}], \tag{4}$$

is a *refinement* of $[\underline{y}, \overline{y}]$ that still contains the system's reachable set. For $\mathcal{H} = \{Hx : x \in \mathbb{R}^n\}$ (with $H \in \mathbb{R}^{m \times n}$ full rank), the tightest refinement is given by the LPs

$$\underline{z}_j = \min_{x \in \mathbb{R}^n} e_j^\top Hx \ \text{ s.t. } \ \underline{y} \leq Hx \leq \overline{y}, \tag{5a}$$

$$\overline{z}_j = \max_{x \in \mathbb{R}^n} e_j^\top Hx \ \text{ s.t. } \ \underline{y} \leq Hx \leq \overline{y}. \tag{5b}$$

In [21], [22], LPs were avoided using a different refinement via dual space sampling, at the cost of looser bounds. Instead, we use `linrax` to embed these LPs directly in the reachability framework, while retaining JIT compilation and automatic differentiation.

In Algorithm 1, interval analysis and refinement as in (4) build the RefinedEmbedding system, whose trajectory $[\underline{y}(t), \overline{y}(t)]$ overapproximates the reachable set as a polytope,

$$\begin{aligned} &\mathcal{R}(t, \{x \in \mathbb{R}^n : \underline{y} \leq Hx \leq \overline{y}\}, [\underline{w}, \overline{w}]) \\ &\qquad \subseteq \{x \in \mathbb{R}^n : \underline{y}(t) \leq Hx \leq \overline{y}(t)\}. \end{aligned}$$

Note that the component replacements on Lines 5-6 and 9-10 *guarantee* the inclusion of degenerate constraints in the generated LP, inhibiting the usage of existing JAX solvers.

B. Safety Filtering Using Automatic Differentiation

We introduce invariance to (2) through the *lifting* matrices

$$H = \begin{bmatrix} \mathbf{I}_n \\ H_2 \end{bmatrix}, \quad H_2 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \end{bmatrix}, \quad H^+ = [\mathbf{I}_n\ 0_{n \times n}],$$

and apply Algorithm 1 to generate refined reachable sets (Figure 1, center). Informally, these matrices "couple" the position states with the heading angle, allowing refinement to exploit their connection through the dynamics. This tightens

Algorithm 1 Safe Control Filtering Using Automatic Differentiation of LPs

Input: full rank $H \in \mathbb{R}^{m\times n}$, left inverse H^+ such that $H^+H = I_n$, initial set $\{x \in \mathbb{R}^n : \underline{y}_0 \le Hx \le \overline{y}_0\}$, initial control map u_{ff}, disturbance set $[\underline{w}, \overline{w}]$, obstacle set $\{x \in \mathbb{R}^n : o(x) \ge 0\}$, time horizon T
function REFINEDEMBEDDING(u_{ff})
  for $i \in \{1, \dots, m\}$ **do**
    $\underline{z} \leftarrow \underline{y}$, $\overline{z} \leftarrow \overline{y}_{i:\underline{y}}$
    $\underline{z}_j \leftarrow \min_x e_j^\top Hx$ s.t. $\underline{y} \le Hx \le \overline{y}_{i:\underline{y}}$, $\forall j$
    $\overline{z}_j \leftarrow \max_x e_j^\top Hx$ s.t. $\underline{y} \le Hx \le \overline{y}_{i:\underline{y}}$, $\forall j$
    $\dot{\underline{y}}_i \leftarrow \min_{y\in[\underline{z},\overline{z}], w\in[\underline{w},\overline{w}]} f_i(H^+y, u_{\text{ff}}(t), w)$
      ▷ *underapproximated using interval analysis*
    $\underline{z} \leftarrow \underline{y}_{i:\overline{y}}$, $\overline{z} \leftarrow \overline{y}$
    $\underline{z}_j \leftarrow \min_x e_j^\top Hx$ s.t. $\underline{y}_{i:\overline{y}} \le Hx \le \overline{y}$, $\forall j$
    $\overline{z}_j \leftarrow \max_x e_j^\top Hx$ s.t. $\underline{y}_{i:\overline{y}} \le Hx \le \overline{y}$, $\forall j$
    $\dot{\overline{y}}_i \leftarrow \max_{y\in[\underline{z},\overline{z}], w\in[\underline{w},\overline{w}]} f_i(H^+y, u_{\text{ff}}(t), w)$
      ▷ *overapproximated using interval analysis*
function SAFETYCHECK(u_{ff})
  $\{[\underline{y}(t), \overline{y}(t)]\}_{t\in[0,T]} \leftarrow$ REFINEDEMBEDDING System Trajectory
  $\overline{\mathcal{R}}_t \leftarrow \{x : \underline{y}(t) \le Hx \le \overline{y}(t)\}$, for every $t \in [0, T]$
  return $\int_0^T \max\{\sup_{x\in\overline{\mathcal{R}}_t} g(x), 0\}\, \mathrm{d}t$
repeat
  $u_{\text{ff}} \leftarrow u_{\text{ff}} - \eta \nabla_{u_{\text{ff}}}$SAFETYCHECK($u_{\text{ff}}$)
until SAFETYCHECK(u_{ff}) ≤ 0
return u_{ff}

the bounds, allowing a small "nudge" to separate them from the obstacle. Thanks to JAX and `linrax`'s automatic differentiation, computing this nudge is straightforward. We quantify the size of $\overline{\mathcal{R}}_t \cap \mathcal{O}$: $\max\{\sup_{x\in\overline{\mathcal{R}}_t} o(x), 0\}$, then sum over time to obtain a function SAFETYCHECK mapping a feedforward input to a number that is positive if the reachable set intersects the obstacle and 0 otherwise. Finally, we apply gradient descent in u_{ff} to modify the input until SAFETYCHECK$(u_{\text{ff}}) = 0$, giving a robustly safe trajectory (Figure 1, right). This example took 8 gradient descent steps in 195s (after 52s of JIT compilation). For now, `linrax` only supports forward-mode autodifferentiation, which contributes to the computation time. Since SAFETYCHECK has many inputs (the nominal control inputs at the sampled time steps) and one output (the safety value), reverse-mode autodifferentiation may be better suited. Though we use Example 1 as a motivating case study, we emphasize that the core computational capabilities of `linrax`—JIT compilation, GPU parallelization, automatic differentiation, and the ability to solve problems with dependent constraints—have a broad scope of control applications.

III. LINEAR PROGRAMS IN JAX

We now describe the general form of LPs and explain the difficulty associated with solving them in JAX. In particular, linearly dependent constraints cannot be handled in the same way as other solvers. We present a modification of the tableau simplex method to address this limitation.

A. Linear Programs

An LP is an optimization problem with a linear objective and linear (equality or inequality) constraints:

$$\min_{x\in[l,u]} c^\top x \text{ s.t. } \begin{matrix} A_{\text{eq}}x = b_{\text{eq}}, \\ A_{\text{ub}}x \le b_{\text{ub}}, \end{matrix} \tag{6}$$

where $x \in \mathbb{R}^n$, $l, u \in \overline{\mathbb{R}}^n$, $l \le u$, $A_{\text{eq}} \in \mathbb{R}^{m_{\text{eq}}\times n}$, $b_{\text{eq}} \in \mathbb{R}^{m_{\text{eq}}}$, $A_{\text{ub}} \in \mathbb{R}^{m_{\text{ub}}\times n}$, $b_{\text{ub}} \in \mathbb{R}^{m_{\text{ub}}}$, and $c \in \mathbb{R}^n$. Most computational solvers specify objective, equality constraints, inequality constraints, and variable bounds separately, as in (6). However, in theory, it suffices to consider a simplification: any such problem can be written in *canonical form*

$$\min_{x\ge 0_n} c^\top x \text{ s.t. } Ax = b, \quad b \ge 0_m, \tag{7}$$

(with larger dimension $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^m$). Inequality constraints are enacted by adding new, non-negative "slack" variables. To minimize over alternative domains, express each input as the difference between two positive components and add constraints as needed.

Converting a problem to canonical form is called *presolving* and may also determine the problem's feasibility or boundedness [23] or eliminate redundant constraints [24]. Pruning dependent constraints not only reduces the problem's dimension, it also ensures that the matrix A is full rank, a necessary condition for many solvers [25], [26]. Unfortunately, this simplification is incompatible with the JAX ecosystem, and will be examined further in Section III-C.

B. The Simplex Method

One common LP solver is the *simplex method*, which we briefly recall here to fix notation (for more details, see [4]). This method iteratively travels between vertices of the feasible polytope, improving the objective at each iteration. When no adjacent vertex is an improvement, an optimal solution has been found and the algorithm terminates. It often runs in two phases; the first phase identifying a vertex of the feasible region which initializes the second phase to solve the original problem. The first phase solves the *auxiliary problem*

$$\min_{x,a\ge 0} \begin{bmatrix} 0_{1\times n} & 1_{1\times m} \end{bmatrix} \begin{bmatrix} x \\ a \end{bmatrix} \text{ s.t. } \begin{bmatrix} A & \mathbf{I}_m \end{bmatrix} \begin{bmatrix} x \\ a \end{bmatrix} = b. \tag{8}$$

Note that $x = 0_{1\times n}$, $a = b$ is an initial feasible vertex of (8). For any optimal (x^*, a^*), if $a^* = 0_{1\times m}$, then x^* is a feasible vertex of (7), as desired. Otherwise, (7) is infeasible.

The state of the simplex method is maintained in a *tableau*. In the first phase, this tableau is initialized to

$$\mathcal{T}_1 = \begin{bmatrix} A & \mathbf{I}_m & b \\ c & 0_{1\times m} & -z \\ -\sum_{i=1}^m A[i,:] & 0_{1\times m} & -w \end{bmatrix}. \tag{9}$$

The upper block represents the constraints, and the bottom two rows give the *reduced cost multipliers* of the original and auxiliary problem. The current feasible vertex is parameterized by a set of *basic variables*; here, those at indices

$n+1, \ldots, n+m$. This structure is maintained throughout every iteration by choosing one variable to enter the set of basic variables and forcing another to exit. The *pivoting strategy* that selects the entering variable is key. In `linrax`, we use *Bland's Rule* [27], a simple approach guaranteed to never cycle. If any index has negative cost multiplier, it may improve the objective when entering the active set; thus, optimization continues until no such index remains.

After choosing an entering variable (c_{enter}), the exiting variable is deduced by the *ratio test*. Using Gauss-Jordan elimination, we cause the column of $x[c_{\text{enter}}]$ to become basic, giving each current basic variable a value of $x[r] = b[r] - A[r, c_{\text{enter}}]x[c_{\text{enter}}]$. To maintain feasibility, $x[c_{\text{enter}}]$ cannot increase past $\frac{b[r]}{A[r,c_{\text{enter}}]}$ for any r where $A[r, c_{\text{enter}}] > 0$. The row which minimizes this ratio is the exiting variable. If no such row exists, then $x[c_{\text{enter}}]$ (and (7)) are unbounded.

After phase 1 is solved, the resulting tableau initializes phase 2. By discarding columns $n+1, \ldots, n+m$ (for the auxiliary variables) and the bottom row (auxiliary cost), a tableau for the original problem is produced. If the problem is *degenerate* (*e.g.*, through linearly dependent constraints), auxiliary variables may remain in the active set, and these variables' rows can also be discarded. (Though theoretically valid, we shall see this is impossible to implement in JAX.) Finally, the pivoting procedure is repeated again until no further improvement is possible, yielding an optimal point.

C. JAX and `linrax`

We now describe `linrax`, a JAX implementation of the simplex method. JAX is a computational toolkit providing *function transformations* that map one function to another. For example, `jacfwd` / `jacrev` return the input's derivative using forward- or reverse-mode automatic differentiation, while `jit` "compiles" the input into equivalent XLA operations. To compute these transformations, JAX first *traces* the input function and represents its computation graph as a `jaxpr`. JAX cannot trace arbitrary Python and restricts the operations allowed in transformable functions [11]. In particular, all arrays in a traceable function must have static shape (*i.e.*, deducible at JIT-compile time). This forbids modifications to the static *shape* of an array based on its traced *values*—a problem for the simplex method.

While the constraint matrix A has static shape, the number of redundant constraints is dependent on its values (specifically its rank), so constraints cannot be pruned in a pre-solve step as is common [7], nor can we discard the auxiliary variables remaining in the active set after phase 1. Non-traceable code that removes dependent constraints can be written, but only to solve LPs in isolation. If solving a LP is merely one part of a pipeline, as is often the case for controls applications (as demonstrated in Section II), then the entire pipeline would no longer benefit from JAX transformations.

To overcome this incompatibility, we introduce Algorithm 2, a modification of the simplex method. We largely follow standard practice, but introduce an extra step before constructing the phase 2 tableau. Since we do not assume full rank A, auxiliary variables may remain in the active set after phase one, and JAX prevents us from discarding these variables as is standard. Instead, we modify their tableau rows so that they harmlessly exit the active set in phase 2. Once they exit, they will not enter again, since (6) does not depend on auxiliary variables. If the problem is feasible, then every auxiliary variable has a value of 0 after phase 1. Therefore, Algorithm 2, Line 19 ensures that whenever an entering variable that would affect the value of any of the remaining auxiliary variables is chosen, that variable receives a ratio of 0 in the ratio test. Since $b \geq 0$ by (7), this is the lowest ratio possible, and these variables exit the active set.

Algorithm 2 The Simplex Method in JAX

Require: Pivot strategy SELECTENTERINGVAR
1: **Input:** Constraint matrix $A \in \mathbb{R}^{m \times n}$, constraint vector $0 \leq b \in \mathbb{R}^m$, objective vector $c \in \mathbb{R}^n$ of the form (7).
2: **function** PIVOT(tableau $\mathcal{T}$, basic indices I)
3: $\quad c_{\text{enter}} \leftarrow$ SELECTENTERINGVAR($\mathcal{T}$)
4: $\quad L \leftarrow \{r : \mathcal{T}[r, c_{\text{enter}}] > 0\}$
5: $\quad$ **if** $L = \emptyset$ **then**
6: $\quad\quad$ **return** The problem is unbounded
7: $\quad r_{\text{exit}} \leftarrow \arg\min_{r \in L} \mathcal{T}[r, -1]/\mathcal{T}[r, c_{\text{enter}}]$
8: $\quad I \leftarrow I \cup \{c_{\text{enter}}\} \setminus \{r_{\text{exit}}\}$
9: $\quad \mathcal{T}[r_{\text{exit}}, :] \leftarrow \mathcal{T}[r_{\text{exit}}, :]/\mathcal{T}[r_{\text{exit}}, c_{\text{enter}}]$
10: $\quad$ **for** $r \in \{1, 2, \ldots, \text{end}\} \setminus \{r_{\text{exit}}\}$ **do**
11: $\quad\quad \mathcal{T}[r, :] \leftarrow \mathcal{T}[r, :] - \mathcal{T}[r, c_{\text{enter}}] \cdot \mathcal{T}[r_{\text{exit}}, :]$
12: $\quad$ **return** $\mathcal{T}$, I
13: $I \leftarrow \{n+1, n+2, \ldots, n+m\}$
14: $\mathcal{T}_1 \leftarrow$ form of (9)
15: **while** Any $\mathcal{T}_1[\text{end}, 1 : \text{end} - 1] < 0$ **do**
16: $\quad \mathcal{T}_1, I \leftarrow$ PIVOT($\mathcal{T}_1$, I)
17: **if** $c_{aux} \cdot x_{aux} > 0$ **then**
18: $\quad$ **return** The problem is infeasible
19: $\mathcal{T}_1[r, :] \leftarrow |\mathcal{T}_1[r, :]|$ for $r \in I, r > n$
20: $\mathcal{T}_2 \leftarrow \texttt{hstack}(\mathcal{T}_1[: \text{end}, : n+1], \mathcal{T}_1[: \text{end}, \text{end}])$
21: **while** Any $\mathcal{T}_2[\text{end}, : \text{end}] < 0$ **do**
22: $\quad$ PIVOT($\mathcal{T}_2$, I)
23: **return** $\mathcal{T}_2$

IV. INTERFACE & COMPARISON

We describe the interface and usage of `linrax` in IV-A, presenting the main function's signature. In IV-B, we compare `linrax` to other LP solvers, including another JAX implementation. On smaller problems (< 50 inputs), its performance is comparable to industry standard toolkits. Additionally, `linrax` is a fully composable JAX subroutine and can be used in large pipelines without sacrificing these advantages, which is not possible with most other solvers.

A. Interface

The signature of the main function in `linrax` is:

```
@partial(jax.jit, static_argnames=["unbounded"])
def linprog(c: jax.Array,
    A_ub: jax.Array, b_ub: jax.Array,
    A_eq: jax.Array, b_eq: jax.Array,
    unbounded: bool
) -> Tuple[SimplexStep,  SimplexSolutionType]:
    ...
```

Here, the first five parameters are as in (6) and closely mimic `scipy`'s interface. Importantly, we do not assume that `A_ub` or `A_eq` are full rank. The last parameter, `unbounded`, determines if minimization is performed on positive inputs or an unbounded domain. Since this change internally doubles the number of input variables, `unbounded` is a *static argument*, so JAX treats it as a constant and re-traces the function whenever it is assigned a different value. This sidesteps the limitation from III-C and allows its value to affect arrays' shapes. Since booleans have only two values, this is a manageable drawback.

The `SimplexSolutionType` object describes the LP. This object contains three **`bool`** fields: `feasible`, `bounded`, and `success`, indicating the corresponding properties of the problem. If minimization was a `success`, the `SimplexStep` object is a solution. The optimal input and objective value are in fields `x` and `fun`, respectively. Furthermore, `tableau` retrieves the full internal tableau, and `basis` gives the indices of the basic variables at `x`.

B. Comparison

We examine `linrax`'s performance relative to several off-the-shelf LP solvers. Both `scipy` [7] and `gurobi` [8] implement specialized algorithms for LPs. These libraries only provide numerical solutions, and do not support any additional features. `CVXPY` [9] solves constrained, convex optimization problems, including LPs, and supports automatic differentiation through the optimization. A library [28] integrates this gradient information with JAX, but is not composable with other transformations such as `jit` or `vmap`. There also exist first-order, JAX-compatible optimizers capable of solving LPs: `JAXopt` [18] and `MPAX` [19][2].

To our knowledge, `linrax` is the first JAX implementation of the simplex method, making it uniquely suited for use as a subroutine in large control pipelines. JAX compatibility provides JIT compilation, vectorization, and GPU parallelization, enabling efficient solving of many LPs simultaneously. Our simplex based algorithm provides an exact solution (up to numerical precision) and works on problems with dependent constraints. This is a clear advantage over existing gradient-based JAX optimizers, which converge to a tolerance (as high as 10^{-4} [19]) and may fail entirely for such problems—even small ones. If the goal is to solve a single LP as fast as possible, other optimizers outperform `linrax` through the use of specialized algorithms, though we are competitive on small problems (Table I). In the more practical case of solving LPs inside a larger computation, `linrax`'s unique benefits become clear (Table II).

All tests were run on a Kubuntu 22.04.05 system, with an Intel Xeon Gold 6320 CPU, Quadro RTX 8000 GPU, and 64GB of RAM. For sample size $N > 0$, each solver optimizes the same problem N times, and we report the mean and standard deviation runtime.

[2]Due to the similarity of `JAXopt`'s solver to that of `MPAX`, and the earlier stage of implementation of the latter, we compare only to the former.

Method	`jit` (sec)	Time (sec)
`scipy`	-	$\mathbf{1.082 \cdot 10^{-3} \pm 7.964 \cdot 10^{-4}}$
`gurobi`	-	$2.23 \cdot 10^{-3} \pm 4.613 \cdot 10^{-3}$
`CVXPY`	-	$1.179 \cdot 10^{-3} \pm 8.879 \cdot 10^{-4}$
`JAXopt`	**0.951**	$3.477 \cdot 10^{-2} \pm 3.497 \cdot 10^{-3}$
`linrax`	1.226	$6.255 \cdot 10^{-3} \pm 7.874 \cdot 10^{-4}$

TABLE I. Comparison of LP solvers on a small, random LP as described in Example 2 Sample size $N = 100$.

Method	`jit` (sec)	Time (sec)	Bound Size
`scipy`	-	37.80653 ± 0.99753	$7.00845 \cdot 10^{-2}$
`gurobi`	-	44.66002 ± 2.20993	$6.8551 \cdot 10^{-2}$
`CVXPY`	-	Fail	-
`JAXopt`	**4.1275**	56.64345 ± 0.57630	$8.84713 \cdot 10^{-2}$
`linrax`	6.25303	$\mathbf{2.13317 \pm 0.054998}$	$6.8724 \cdot 10^{-2}$

TABLE II. Comparison of LP solvers on the interval refinement problem Example 3 with $\mu = 1$, $t_f = 0.628$. Sample size $N = 10$. Bound size is quantified as the product of the widths of the interval bound for the reachable set; smaller sizes mean less conservatism.

Example 2 (Random, small linear program)**.** A random LP with $n = 20$ inputs and $m_{ub} = 15$ inequality constraints is generated and optimized by each solver.

The results of this test are given in Table I: `linrax` exhibits performance on the same order of magnitude as professional solvers, beating the other JAX implementation.

In the second test, we use each solver to compute an over-approximation of a dynamical system's reachable set.

Example 3 (Reachable Set Approximation)**.** Consider the Van der Pol oscillator, described by the ODE

$$\dot{x}_1 = \mu \left(x_1 - \frac{1}{3} x_1^3 - x_2 \right), \quad \dot{x}_2 = \frac{x_1}{\mu}, \tag{10}$$

for some $\mu > 0$. Given an initial condition $x(0)$, let $\phi(t, x(0))$ be the solution to (10). For $t_f > 0$, we use the approach from II-A with the JAX interval analysis toolbox `immrax` [12] to compute a polytope containing $\phi(t_f, x(0))$ for any initial condition inside a given box $x(0) \in [\underline{x}_0, \overline{x}_0]$.

Example 3 requires the frequent solving of small LPs as in Algorithm 1 (where each LP is automatically generated and contains dependent constraints). A plot of the reachable sets generated by our approach is given in Figure 2, and results are summarized in Table II. Here, `linrax` excels, an order of magnitude faster than any other implementation. In fact, we are *still* the fastest even when including JIT compile time, showing that the performance gains more than outweigh the oft-cited drawback of compilation times. This is due to the extra benefits of embedding the LP directly into the reachable set pipeline, allowing us to `jit` the entire procedure. The two gradient-based libraries perform especially poorly, with `CVXPY` crashing before the test is complete and `JAXopt` failing to converge to its default tolerance before the iteration limit for many programs, resulting in the slowest time and significantly worse final bound size.

V. Conclusions

We have presented the first simplex-based LP solver compatible with the JAX ecosystem. This work was motivated by

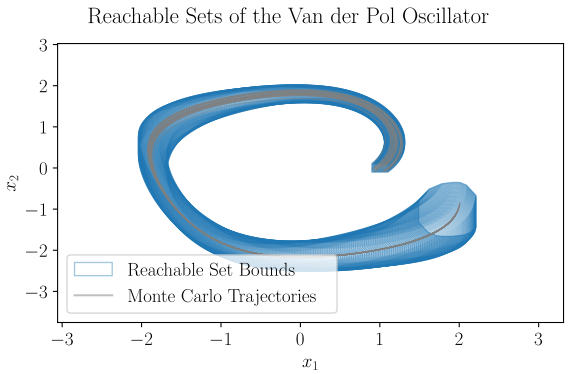

Fig. 2. Bounds on the reachable set of trajectories of the Van der Pol oscillator with $\mu = 1$, $t_f = 2\pi$, $x_1(0) \in [0.9, 1.1]$, and $x_2(0) \in [-0.1, 0.1]$, computed with a LP-based interval refinement procedure. Real Monte-Carlo sampled trajectories are shown in gray as a "ground-truth" comparison to these bounds. Note how the bounds always contain every sampled trajectory, and remain relatively tight over a long time horizon.

two complementary trends: the increased use of optimization as a subroutine in control algorithms, and the growing popularity of JAX in controls applications. Stringent JAX requirements prevent direct implementation of standard simplex methods, particularly the handling of dependent constraints, a necessity in many algorithms such as the reachability-based safety control problem described in Section II. To overcome this limitation, we introduce a novel modification ensuring full compatibility with JAX requirements. We showed that when embedded into a JAX pipeline, `linrax` outperforms state-of-the-art solvers, and demonstrated an application of automatic differentiation to solve safe robotics and controls problems. An interesting possibility for future work would be to implement improvements to the simplex method, such as parallelization [25], or a more sophisticated pivoting rule.

References

[1] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "CasADi: a software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, vol. 11, pp. 1–36, Mar. 2019.

[2] E. D. Sontag, "Control-Lyapunov functions," in *Open Problems in Mathematical Systems and Control Theory* (V. Blondel, E. D. Sontag, M. Vidyasagar, and J. C. Willems, eds.), pp. 211–216, London: Springer, 1999.

[3] A. D. Ames, S. Coogan, M. Egerstedt, G. Notomista, K. Sreenath, and P. Tabuada, "Control Barrier Functions: Theory and Applications," in *2019 18th European Control Conference (ECC)*, pp. 3420–3431, June 2019.

[4] G. Dantzig, *Linear Programming and Extensions*. Princeton Landmarks in Mathematics and Physics, Princeton, NJ: Princeton University Press, 2016.

[5] M. Bahreinian, E. Aasi, and R. Tron, "Robust Path Planning and Control For Polygonal Environments via Linear Programming," in *2021 American Control Conference (ACC)*, pp. 5035–5042, May 2021.

[6] S. M. Harwood and P. I. Barton, "Efficient polyhedral enclosures for the reachable set of nonlinear control systems," *Springer London*, Feb. 2016.

[7] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, I. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental algorithms for scientific computing in python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[8] Gurobi Optimization, LLC, "Gurobi optimizer reference manual," 2024.

[9] S. Diamond and S. Boyd, "CVXPY: A Python-embedded modeling language for convex optimization," *Journal of Machine Learning Research*, vol. 17, no. 83, pp. 1–5, 2016.

[10] D. G. Spampinato and A. C. Elster, "Linear optimization on modern GPUs," in *2009 IEEE international symposium on parallel & distributed processing*, pp. 1–8, IEEE, 2009.

[11] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018.

[12] A. Harapanahalli, S. Jafarpour, and S. Coogan, "immrax: A Parallelizable and Differentiable Toolbox for Interval Analysis and Mixed Monotone Reachability in JAX," Apr. 2024.

[13] L. Grillotti and A. Cully, "Kheperax: a Lightweight JAX-based Robot Control Environment for Benchmarking Quality-Diversity Algorithms," in *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, GECCO '23 Companion, (New York, NY, USA), Association for Computing Machinery, July 2023.

[14] A. Rutherford, B. Ellis, M. Gallici, J. Cook, A. Lupu, G. Ingvarsson, T. Willi, R. Hammond, A. Khan, C. S. de Witt, A. Souly, S. Bandyopadhyay, M. Samvelyan, M. Jiang, R. Lange, S. Whiteson, B. Lacerda, N. Hawes, T. Rocktäschel, C. Lu, and J. Foerster, "JaxMARL: Multi-Agent RL Environments and Algorithms in JAX," *Advances in Neural Information Processing Systems*, vol. 37, Dec. 2024.

[15] DeepMind, I. Babuschkin, K. Baumli, A. Bell, S. Bhupatiraju, J. Bruce, P. Buchlovsky, D. Budden, T. Cai, A. Clark, I. Danihelka, A. Dedieu, C. Fantacci, J. Godwin, C. Jones, R. Hemsley, T. Hennigan, M. Hessel, S. Hou, S. Kapturowski, T. Keck, I. Kemaev, M. King, M. Kunesch, L. Martens, H. Merzic, V. Mikulik, T. Norman, G. Papamakarios, J. Quan, R. Ring, F. Ruiz, A. Sanchez, L. Sartran, R. Schneider, E. Sezener, S. Spencer, S. Srinivasan, M. Stanojević, W. Stokowiec, L. Wang, G. Zhou, and F. Viola, "The DeepMind JAX Ecosystem," 2020.

[16] J. Rader, T. Lyons, and P. Kidger, "Optimistix: modular optimisation in JAX and equinox," *arXiv:2402.09983*, 2024.

[17] P. Kidger and C. Garcia, "Equinox: neural networks in JAX via callable PyTrees and filtered transformations," *Differentiable Programming workshop at Neural Information Processing Systems 2021*, 2021.

[18] M. Blondel, Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert, "Efficient and modular implicit differentiation," *arXiv preprint arXiv:2105.15183*, 2021.

[19] H. Lu, Z. Peng, and J. Yang, "MPAX: Mathematical programming in JAX," *arXiv preprint arXiv:2412.09734*, 2024.

[20] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter, *Applied interval analysis*. Springer, 1st edition. ed., 2001.

[21] K. Shen and J. Scott, "Rapid and Accurate Reachability Analysis for Nonlinear Dynamic Systems by Exploiting Model Redundancy," *Computers & Chemical Engineering*, vol. 106, Aug. 2017.

[22] A. Harapanahalli and S. Coogan, "Certified Robust Invariant Polytope Training in Neural Controlled ODEs," Aug. 2024.

[23] E. D. Andersen and K. D. Andersen, "Presolving in linear programming," *Mathematical Programming*, vol. 71, pp. 221–245, Dec. 1995.

[24] E. D. Andersen, "Finding all linearly dependent rows in large-scale linear programming," *Optimization Methods and Software*, vol. 6, pp. 219–227, Jan. 1995.

[25] Q. Huangfu and J. A. J. Hall, "Parallelizing the dual revised simplex method," *Mathematical Programming Computation*, vol. 10, Mar. 2018.

[26] R. Nishihara, L. Lessard, B. Recht, A. Packard, and M. Jordan, "A general analysis of the convergence of ADMM," in *Proceedings of the 32nd international conference on machine learning* (F. Bach and D. Blei, eds.), vol. 37 of *Proceedings of machine learning research*, (Lille, France), pp. 343–352, PMLR, July 2015.

[27] D. Avis and V. Chvátal, "Notes on Bland's pivoting rule," in *Polyhedral Combinatorics: Dedicated to the memory of D.R. Fulkerson* (M. L. Balinski and A. J. Hoffman, eds.), pp. 24–34, Berlin, Heidelberg: Springer, 1978.

[28] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and J. Z. Kolter, "Differentiable convex optimization layers," in *Advances in neural information processing systems*, pp. 9558–9570, 2019.