DOI: 10.14529/jsfi240301

AlFaMove: Scalable Implementation of Surface Movement Method for Cluster Computing Systems*

Nikolay A. Olkhovsky[1], ***Leonid B. Sokolinsky***[1]

The article presents a numerical implementation of the surface movement method for linear programming. The base of this implementation is the new AlFaMove algorithm, which builds on the surface of a feasible polytope an optimal objective path from an arbitrary boundary point to a point that is a solution to a linear programming problem. The optimal objective path is a path along the faces of the feasible polytope in the direction of maximizing the value of the objective function. To calculate the optimal movement direction, the pseudoprojection operation on a linear manifold is used. The pseudoprojection operation is a generalization of the orthogonal projection and is implemented using an iterative projection-type algorithm. The proposition is proved that, for a linear manifold that is the intersection of hyperplanes, the pseudoprojection coincides with the orthogonal projection. It is also proved that, in the case of a linear manifold, pseudoprojection makes it possible to calculate the movement vector in the direction of maximum increase of the objective function. A parallel implementation of the AlFaMove algorithm is described. The results of computational experiments on a cluster computing system are presented to demonstrate the high scalability of the proposed numerical implementation.

Keywords: linear programming, surface movement method, numerical implementation, AlFaMove algorithm, parallel implementation, cluster computing system, scalability evaluation.

Introduction

The age of big data and Industry 4.0 generated large-scale linear programming (LP) problems including millions of variables and millions of constraints [4, 8, 12, 13]. In many cases, the object of linear programming is problems related to the optimization of non-stationary processes [2]. In non-stationary LP problems, the objective function and/or constraints change during the computational process. Also in this class of problems, there are applications in which it is necessary to perform optimization in real time. Highly scalable methods and parallel algorithms for linear programming are needed to solve such problems.

The simplest approach to solving non-stationary optimization problems is to consider each change as the appearance of a new optimization problem that needs to be solved from scratch [2]. However, this approach is often impractical, because solving a problem from scratch without reusing information from the past can take too long. Thus, it is desirable to have an optimization algorithm capable of continuously adapting the computation process to a changing environment, reusing information obtained in the past. This approach is applicable for real-time processes if the algorithm tracks the trajectory of the optimal point fast enough. In the case of large-scale LP problems, the latter requires the development of scalable methods and parallel algorithms for linear programming.

To date, the most popular methods for solving LP problems are the simplex method [3] and the interior-point methods [20]. These methods are capable of solving problems with tens of thousands of variables and constraints. However, the scalability of parallel algorithms based on the simplex method, in general case, is limited to 16–32 processor nodes [10]. As regards the interior-point algorithms, in general case, they are not amenable to effective parallelization. This

*The paper is recommended for publication by the Program Committee of the International Scientific Conference "Russian Supercomputing Days 2024".

[1]South Ural State University (National Research University), Chelyabinsk, Russian Federation

limits the use of these methods for solving large-scale non-stationary LP problems in real time. In accordance with this, the task of developing scalable methods and efficient parallel algorithms for linear programming on cluster computing systems remains urgent.

In recent paper [11] a theoretical description of the new surface movement method for linear programming was presented. This method builds an optimal objective path on the surface of the feasible polytope[2] from an arbitrary boundary point to a solution of the LP problem. The optimal objective path is a path along the faces of the feasible polytope in the direction of maximizing the value of the objective function. Algorithm 1 proposed in this paper, in Step 15, requires finding a point with the maximum value of the objective function on the boundary of a hyperdisk. At the same time, the paper does not provide a numerical algorithm that allows you to perform this step. In this article, we present and evaluate the AlFaMove algorithm, which eliminates the gap. The rest of the paper is organized as follows. Section 1 presents the theoretical background on which the surface movement method and AlFaMove algorithm are based. Section 2 is devoted to the description of the pseudoprojection operation, which allows you to find a movement vector along the optimal objective path for a linear manifold resulting from the intersection of hyperplanes. Section 3 provides a formalized description of the AlFaMove algorithm, which is a numerical implementation of the surface movement method. Section 4 describes a parallel version of the AlFaMove algorithm. Section 5 provides information on the software implementation of the AlFaMove algorithm and the results of experiments on a cluster computing system to evaluate its scalability. Conclusion summarizes the results and provides further research directions.

1. Theoretical Background

This section contains the necessary theoretical basis used to describe the AlFaMove algorithm. We consider a LP problem in the following form:

$$\bar{\boldsymbol{x}} = \arg\max_{\boldsymbol{x}\in\mathbb{R}^n} \{\langle \boldsymbol{c}, \boldsymbol{x}\rangle \,|\, A\boldsymbol{x} \leqslant \boldsymbol{b}\}, \tag{1}$$

where $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$, $A \in \mathbb{R}^{m\times n}$, $m > 1$, $\boldsymbol{c} \neq \boldsymbol{0}$. Here, $\langle\cdot,\cdot\rangle$ stands for the dot product of two vectors. We assume that the constraint $\boldsymbol{x} \geqslant \boldsymbol{0}$ is also included in the matrix inequality $A\boldsymbol{x} \leqslant \boldsymbol{b}$ in the form of $-\boldsymbol{x} \leqslant \boldsymbol{0}$. The linear objective function of the problem (1) has the form

$$f(\boldsymbol{x}) = \langle \boldsymbol{c}, \boldsymbol{x}\rangle .$$

In this case, the vector $\boldsymbol{c}$ is the gradient of the objective function $f(\boldsymbol{x})$.

Let $\boldsymbol{a}_i \in \mathbb{R}^n$ denote a vector representing the ith row of the matrix A. We assume that $\boldsymbol{a}_i \neq \boldsymbol{0}$ for all $i \in \{1,\ldots,m\}$. Denote by $\hat{H}_i$ a closed half-space defined by the inequality $\langle \boldsymbol{a}_i, \boldsymbol{x}\rangle \leqslant b_i$, and by H_i – the hyperplane bounding it:

$$\hat{H}_i = \{\boldsymbol{x} \in \mathbb{R}^n | \langle \boldsymbol{a}_i, \boldsymbol{x}\rangle \leqslant b_i\}; \tag{2}$$

$$H_i = \{\boldsymbol{x} \in \mathbb{R}^n | \langle \boldsymbol{a}_i, \boldsymbol{x}\rangle = b_i\}. \tag{3}$$

Let us define a feasible polytope

$$M = \bigcap_{i\in\mathcal{P}} \hat{H}_i, \tag{4}$$

[2]The feasible polytope is the feasible region of the LP problem.

representing the feasible region of LP Problem (1). Note that M, in this case, is a closed convex set. We assume that M is bounded, and $M \neq \emptyset$, i.e., LP Problem (1) has a solution.

Let us define a recessive half-space [17].

Definition 1. The half-space $\hat{H}_i$ is called recessive if

$$\forall \boldsymbol{x} \in H_i, \forall \lambda > 0 : \boldsymbol{x} + \lambda \boldsymbol{c} \notin \hat{H}_i. \tag{5}$$

The geometric meaning of this definition is that a ray outgoing in the direction of the vector $\boldsymbol{c}$ from any point of the hyperplane bounding the recessive half-space has no points in common with this half-space, except for the beginning one. It is known [17] that the following condition is necessary and sufficient for the half-space $\hat{H}_i$ to be recessive:

$$\langle \boldsymbol{a}_i, \boldsymbol{c} \rangle > 0.$$

Denote

$$\mathcal{I} = \{ i \in \{1, \ldots, m\} \,|\langle \boldsymbol{a}_i, \boldsymbol{c} \rangle > 0 \}, \tag{6}$$

i.e., $\mathcal{I}$ represents a set of indexes for which the half-space $\hat{H}_i$ is recessive. Since the feasible polytope M is a bounded set, we have

$$\mathcal{I} \neq \emptyset.$$

Define

$$\hat{M} = \bigcap_{i \in \mathcal{I}} \hat{H}_i. \tag{7}$$

Obviously, $\hat{M}$ is a convex, closed, unbounded polytope. We will call it recessive. Let us denote by $\Gamma(M)$ the set of boundary points of the feasible polytope M, and by $\Gamma(\hat{M})$ the set of boundary points of the recessive polytope $\hat{M}$[3]. According to Proposition 3 in [17] we have

$$\bar{\boldsymbol{x}} \in \Gamma(\hat{M}),$$

i.e., a solution to LP problem (1) lies on the boundary of the recessive polytope $\hat{M}$.

Following [9], we can define an orthogonal projection onto a hyperplane.

Definition 2. Let be given the hyperplane $H = \{\boldsymbol{x} \in \mathbb{R}^n | \langle \boldsymbol{a}, \boldsymbol{x} \rangle = b\}$. The orthogonal projection $\boldsymbol{\pi}_H(\boldsymbol{v})$ of a point $\boldsymbol{v} \in \mathbb{R}^n$ onto the hyperplane H is defined by the equation

$$\boldsymbol{\pi}_H(\boldsymbol{v}) = \boldsymbol{v} - \frac{\langle \boldsymbol{a}, \boldsymbol{v} \rangle - b}{\|\boldsymbol{a}\|^2} \boldsymbol{a}. \tag{8}$$

The following proposition provides a way to calculate the optimal path on a hyperplane.

Proposition 1. Let be given a hyperplane H with the normal $\boldsymbol{a} \in \mathbb{R}^n$, which including the point $\boldsymbol{u} \in \mathbb{R}^n$:

$$H = \{ \boldsymbol{x} \in \mathbb{R}^n \,|\langle \boldsymbol{a}, \boldsymbol{x} \rangle = \langle \boldsymbol{a}, \boldsymbol{u} \rangle \}. \tag{9}$$

Let a linear function $f(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}$ with gradient $\boldsymbol{c} \in \mathbb{R}^n$ be defined:

$$f(\boldsymbol{x}) = \langle \boldsymbol{c}, \boldsymbol{x} \rangle. \tag{10}$$

[3] A boundary point of a set $\hat{M} \subset \mathbb{R}^n$ is a point in $\mathbb{R}^n$ for which any open neighborhood of it in $\mathbb{R}^n$ has a nonempty intersection with both the set $\hat{M}$ and its complement.

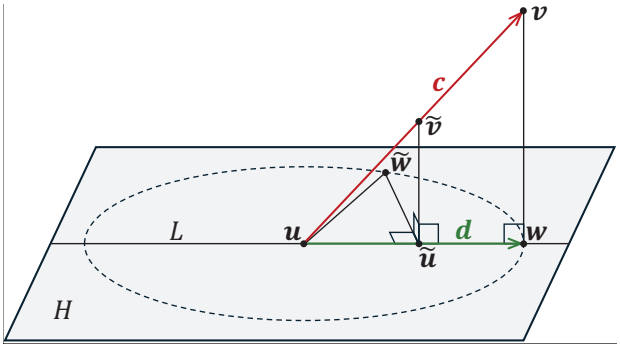

Figure 1. Illustration to proof of Proposition 1
(the dashed line denotes the hyper-circle of radius $\|\boldsymbol{w}-\boldsymbol{u}\|$ centered at the point $\boldsymbol{u}$)

Let the vectors $\boldsymbol{a}$ and $\boldsymbol{c}$ be linearly independent (not collinear, and there is no zero vector among them). Denote

$$\boldsymbol{v} = \boldsymbol{u} + \boldsymbol{c}. \tag{11}$$

Build the orthogonal projection $\boldsymbol{\pi}_H(\boldsymbol{v})$ of point $\boldsymbol{v}$ onto the hyperplane H:

$$\boldsymbol{w} = \boldsymbol{\pi}_H(\boldsymbol{v}). \tag{12}$$

Then the vector $\boldsymbol{d} = \boldsymbol{w} - \boldsymbol{u}$ uniquely determines the direction of maximum increase of the linear function $f(\boldsymbol{x})$ defined by equation (10).

Proof. Assume the opposite is true: there exists a point $\tilde{\boldsymbol{w}} \in H$ such that

$$\langle \boldsymbol{c}, \tilde{\boldsymbol{w}} \rangle \geqslant \langle \boldsymbol{c}, \boldsymbol{w} \rangle, \tag{13}$$

$\|\tilde{\boldsymbol{w}} - \boldsymbol{u}\| = \|\boldsymbol{w} - \boldsymbol{u}\|$, and $\tilde{\boldsymbol{w}} \neq \boldsymbol{w}$ (see Fig. 1). Here and further on, $\|\cdot\|$ denotes the Euclidean norm. Calculate $\langle \boldsymbol{c}, \boldsymbol{w} \rangle$. According to the definition 2, the orthogonal projection $\boldsymbol{\pi}_H(\boldsymbol{v})$ of point $\boldsymbol{v}$ onto the hyperplane H defined by equation (9) is calculated as follows:

$$\boldsymbol{w} = \boldsymbol{v} - \frac{\langle \boldsymbol{a}, \boldsymbol{v} - \boldsymbol{u} \rangle}{\|\boldsymbol{a}\|^2} \boldsymbol{a}.$$

Substituting the right side of equation (11) instead of $\boldsymbol{v}$, we obtain

$$\boldsymbol{w} = \boldsymbol{u} + \boldsymbol{c} - \frac{\langle \boldsymbol{a}, \boldsymbol{c} \rangle}{\|\boldsymbol{a}\|^2} \boldsymbol{a}. \tag{14}$$

Using (14), we figure out

$$\langle \boldsymbol{c}, \boldsymbol{w} \rangle = \langle \boldsymbol{c}, \boldsymbol{u} \rangle + \|\boldsymbol{c}\|^2 - \frac{\langle \boldsymbol{a}, \boldsymbol{c} \rangle^2}{\|\boldsymbol{a}\|^2}. \tag{15}$$

Since $\boldsymbol{a}$ and $\boldsymbol{c}$ are linearly independent, in accordance with the Cauchy–Bunyakovsky–Schwarz inequality we have

$$\langle \boldsymbol{a}, \boldsymbol{c} \rangle^2 < \|\boldsymbol{a}\|^2 \cdot \|\boldsymbol{c}\|^2.$$

This implies

$$\|\boldsymbol{c}\|^2 - \frac{\langle \boldsymbol{a}, \boldsymbol{c} \rangle^2}{\|\boldsymbol{a}\|^2} > 0. \tag{16}$$

Now calculate $\langle \boldsymbol{c}, \tilde{\boldsymbol{w}} \rangle$. Let $\tilde{\boldsymbol{u}} = \boldsymbol{\pi_L}(\tilde{\boldsymbol{w}})$ be the orthogonal projection of point $\tilde{\boldsymbol{w}}$ onto the line L passing through the points $\boldsymbol{u}$ and $\boldsymbol{w}$. By construction, there is a number n satisfying the condition

$$-1 \leqslant \delta < 1 \tag{17}$$

such that

$$\tilde{\boldsymbol{u}} = \boldsymbol{u} + \delta(\boldsymbol{w} - \boldsymbol{u}).$$

Define

$$\tilde{\boldsymbol{v}} = \boldsymbol{u} + \delta(\boldsymbol{v} - \boldsymbol{u}). \tag{18}$$

Then, the point $\tilde{\boldsymbol{u}}$ is the orthogonal projection of the point $\tilde{\boldsymbol{v}}$ onto the hyperplane H defined by equation (9), and can be calculated as follows:

$$\tilde{\boldsymbol{u}} = \tilde{\boldsymbol{v}} - \frac{\langle \boldsymbol{a}, \tilde{\boldsymbol{v}} - \boldsymbol{u} \rangle}{\|\boldsymbol{a}\|^2} \boldsymbol{a}.$$

Substituting the right side of equation (18) instead of $\tilde{\boldsymbol{v}}$, we obtain the following equation from here:

$$\tilde{\boldsymbol{u}} = \boldsymbol{u} + \delta \left(\boldsymbol{v} - \boldsymbol{u} - \frac{\langle \boldsymbol{a}, \boldsymbol{v} - \boldsymbol{u} \rangle}{\|\boldsymbol{a}\|^2} \boldsymbol{a} \right).$$

Using (11), we make the replacement $\boldsymbol{v} - \boldsymbol{u} = \boldsymbol{c}$:

$$\tilde{\boldsymbol{u}} = \boldsymbol{u} + \delta \left(\boldsymbol{c} - \frac{\langle \boldsymbol{a}, \boldsymbol{c} \rangle}{\|\boldsymbol{a}\|^2} \boldsymbol{a} \right). \tag{19}$$

Obviously

$$\tilde{\boldsymbol{w}} = (\tilde{\boldsymbol{w}} - \tilde{\boldsymbol{u}}) + \tilde{\boldsymbol{u}}. \tag{20}$$

Replace the second summand in (20) with the right-hand side of equation (19):

$$\tilde{\boldsymbol{w}} = (\tilde{\boldsymbol{w}} - \tilde{\boldsymbol{u}}) + \boldsymbol{u} + \delta \left(\boldsymbol{c} - \frac{\langle \boldsymbol{a}, \boldsymbol{c} \rangle}{\|\boldsymbol{a}\|^2} \boldsymbol{a} \right).$$

Using (1), we obtain

$$\langle \boldsymbol{c}, \tilde{\boldsymbol{w}} \rangle = \langle \boldsymbol{c}, \tilde{\boldsymbol{w}} - \tilde{\boldsymbol{u}} \rangle + \langle \boldsymbol{c}, \boldsymbol{u} \rangle + \delta \left(\|\boldsymbol{c}\|^2 - \frac{\langle \boldsymbol{a}, \boldsymbol{c} \rangle^2}{\|\boldsymbol{a}\|^2} \right).$$

By construction, vector $\tilde{\boldsymbol{w}} - \tilde{\boldsymbol{u}}$ is orthogonal to vector $\boldsymbol{v} - \boldsymbol{u} = \boldsymbol{c}$. Therefore, $\langle \boldsymbol{c}, \tilde{\boldsymbol{w}} - \tilde{\boldsymbol{u}} \rangle = 0$. Thus, equation (1) is transformed to the form

$$\langle \boldsymbol{c}, \tilde{\boldsymbol{w}} \rangle = \langle \boldsymbol{c}, \boldsymbol{u} \rangle + \delta \left(\|\boldsymbol{c}\|^2 - \frac{\langle \boldsymbol{a}, \boldsymbol{c} \rangle^2}{\|\boldsymbol{a}\|^2} \right). \tag{21}$$

Comparing (15) and (21), and taking into account (16) and (17), we obtain

$$\langle \boldsymbol{c}, \tilde{\boldsymbol{w}} \rangle < \langle \boldsymbol{c}, \boldsymbol{w} \rangle,$$

which contradicts(13). □

Returning to the LP (1) problem, we can say the following. Let $\boldsymbol{u} \in M \cap \Gamma(\hat{M})$, and there is a single recessive hyperplane $H_{i'}$ $(i' \in \mathcal{I})$ such that $\boldsymbol{u} \in H_{i'}$. In this case, the vector $\boldsymbol{d}$, which determines the direction of the optimal objective path at the point $\boldsymbol{u}$, in accordance with Proposition 1, can be calculated as follows:

$$\boldsymbol{d} = \boldsymbol{c} - \frac{\langle \boldsymbol{a}_{i'}, \boldsymbol{u} + \boldsymbol{c} \rangle - b_{i'}}{\|\boldsymbol{a}_{i'}\|^2} \boldsymbol{a}_{i'}. \tag{22}$$

In the next section, we consider the case when two or more hyperplanes pass through the point $\boldsymbol{u}$.

2. Pseudoprojecting onto Linear Manifold

Let $\mathcal{J} \subseteq \{1, \ldots, m\}$, $\mathcal{J} \neq \emptyset$, and $\bigcap\limits_{i \in \mathcal{J}} H_i \neq \emptyset$. In this case, the set of indices $\mathcal{J}$ defines a linear manifold L in the space $\mathbb{R}^n$:

$$L = \bigcap_{i \in \mathcal{J}} H_i. \tag{23}$$

Denote by k_L the dimension of the linear manifold L. For $0 < k_L < n-1$, the manifold L is not a hyperplane, and, to determine the movement vector $\boldsymbol{d}$ along this manifold in the direction of maximum increase in the objective function value, the equation (22) cannot be used, since such a linear manifold cannot be defined by a single linear equation in the space $\mathbb{R}^n$. However, we can find the specified vector $\boldsymbol{d}$ using the pseudoprojection operation [17]. Define the projection mapping $\boldsymbol{\varphi}(\cdot)$:

$$\boldsymbol{\varphi}(\boldsymbol{x}) = \frac{1}{|\mathcal{J}|} \sum_{i \in \mathcal{J}} \boldsymbol{\pi}_{H_i}(\boldsymbol{x}). \tag{24}$$

It is known [18] that the mapping $\boldsymbol{\varphi}(\boldsymbol{x})$ is a continuous L-Fejér mapping, and the sequence of points

$$\left\{ \boldsymbol{x}_k = \boldsymbol{\varphi}^k(\boldsymbol{x}_0) \right\}_{k=1}^{\infty} \tag{25}$$

generated by this mapping converges to a point belonging to L:

$$\boldsymbol{x}_k \to \tilde{\boldsymbol{x}} \in L.$$

Using the mapping $\boldsymbol{\varphi}(\cdot)$, let us define the pseudoprojection on a linear manifold formed by the intersection of hyperplanes.

Definition 3. Let $\mathcal{J} \subseteq \{1, \ldots, m\}$, $\mathcal{J} \neq \emptyset$, $\bigcap_{i \in \mathcal{J}} H_i \neq \emptyset$, and $\boldsymbol{\varphi}(\cdot)$ be the projection mapping defined by equation (24). The pseudoprojection $\boldsymbol{\rho}_{\mathcal{J}}(\boldsymbol{x})$ of the point $\boldsymbol{x} \in \mathbb{R}^n$ onto the linear manifold $L = \bigcap_{i \in \mathcal{J}} H_i$ is the limit point of the sequence (25):

$$\lim_{k \to \infty} \left\| \boldsymbol{\rho}_{\mathcal{J}}(\boldsymbol{x}) - \boldsymbol{\varphi}^k(\boldsymbol{x}) \right\| = 0.$$

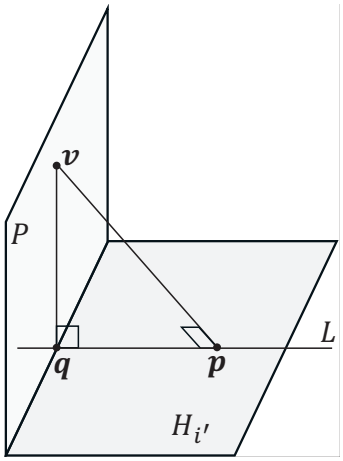

Figure 2. Illustration to proof of Lemma 1

An important feature of the pseudoprojection onto a linear manifold is that the pseudoprojection coincides with the orthogonal projection in this case. To prove this fact, we need the following lemma.

Lemma 1. Let the hyperplane $H_{i'}$ and the linear manifold L belonging to this hyperplane be given in the space $\mathbb{R}^n$:

$$H_{i'} = \{\boldsymbol{x} \in \mathbb{R}^n | \, \langle \boldsymbol{a}_{i'}, \boldsymbol{x} \rangle = b_{i'}\};$$
$$L = \bigcap\nolimits_{i \in \mathcal{J}} H_i;$$
$$i' \in \mathcal{J}.$$

Denote by P the linear manifold that is the orthogonal complement to L:

$$P = L^{\perp}. \tag{26}$$

Then for any point $\boldsymbol{v}$ belonging to the linear manifold P, its orthogonal projection $\boldsymbol{\pi}_{H_{i'}}(\boldsymbol{v})$ onto the hyperplane $H_{i'}$ also belongs to the linear manifold P:

$$\forall \boldsymbol{v} \in P : \boldsymbol{\pi}_{H_{i'}}(\boldsymbol{v}) \in P.$$

Proof. Denote by $\boldsymbol{p}$ the orthogonal projection of the point $\boldsymbol{v}$ onto the hyperplane $H_{i'}$:

$$\boldsymbol{p} = \boldsymbol{\pi}_{H_{i'}}(\boldsymbol{v}). \tag{27}$$

Without loss of generality, we can assume that $\boldsymbol{p} \in L$ (see Fig. 2). Suppose that the point $\boldsymbol{p}$ does not belong to the linear manifold P. Let us by $\boldsymbol{q}$ denote the intersection point of a linear manifold L with its orthogonal complement P:

$$\boldsymbol{q} = L \cap P.$$

Since P is the orthogonal complement to the linear manifold L, the point $\boldsymbol{q}$ is the orthogonal projection of the point $\boldsymbol{p}$ onto the linear manifold P:

$$\boldsymbol{q} = \boldsymbol{\pi}_P(\boldsymbol{p}). \tag{28}$$

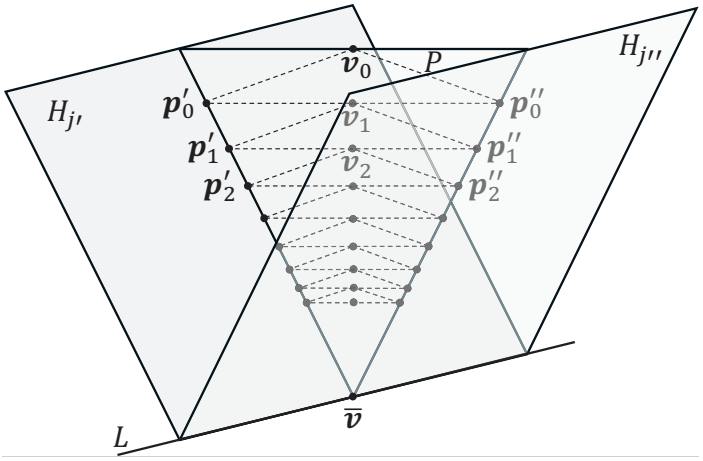

Figure 3. Illustration to proof of Proposition 2 for $n = 3$

Consider the triangle $\triangle(\boldsymbol{v}, \boldsymbol{p}, \boldsymbol{q})$. By virtue of (27), the angle $\angle\boldsymbol{p}$ with the vertex at the point $\boldsymbol{p}$ is right. But this is only possible if $\boldsymbol{p} = \boldsymbol{q}$, that is $\boldsymbol{p} \in P$. □

The following proposition proves that a pseudoprojection on a linear manifold coincides with an orthogonal projection.

Proposition 2. Let the following conditions hold:

$$\mathcal{J} \subseteq \{1, \ldots, m\}, \tag{29}$$

$$L = \bigcap_{i \in \mathcal{J}} H_i,\ L \neq \emptyset; \tag{30}$$

where $H_i = \{\boldsymbol{x} \in \mathbb{R}^n | \langle \boldsymbol{a}_i, \boldsymbol{x} \rangle = b_i\}$. Denote by $\boldsymbol{\pi}_L(\boldsymbol{x})$ the orthogonal projection of the point $\boldsymbol{x} \in \mathbb{R}^n$ onto the linear manifold L. Then,

$$\boldsymbol{\rho}_L(\boldsymbol{x}) = \boldsymbol{\pi}_L(\boldsymbol{x}),$$

i.e., the pseudoprojection onto the linear manifold L coincides with the orthogonal projection.

Proof. Fix an arbitrary point $\boldsymbol{v}_0 \in \mathbb{R}^n$. Consider the linear manifold P containing the point $\boldsymbol{v}_0$ and being the orthogonal complement to the linear manifold L:

$$\boldsymbol{v}_0 \in P = L^{\perp}. \tag{31}$$

Denote by $\bar{\boldsymbol{v}}$ the intersection point of the linear manifold L with its orthogonal complement P:

$$L \cap P = \{\bar{\boldsymbol{v}}\}$$

(see Fig. 3). Make the orthogonal projection of the point $\boldsymbol{v}_0$ onto the hyperplane H_j for an arbitrary $j \in \mathcal{J}$:

$$\boldsymbol{p}_0 = \boldsymbol{\pi}_{H_j}(\boldsymbol{v}_0).$$

According to Lemma 1, we have

$$\boldsymbol{\pi}_{H_j}(\boldsymbol{v}_0) \in P.$$

It follows from this and from (24) that

$$\boldsymbol{v}_1 = \varphi(\boldsymbol{v}_0) \in P.$$

This means that the sequence

$$\left\{\boldsymbol{v}_k = \boldsymbol{\varphi}^k\left(\boldsymbol{v}_0\right)\right\}_{k=1}^{\infty}$$

converges to the point $\bar{\boldsymbol{v}}$ of the intersection of the linear manifold L with the linear manifold P, i.e., $\boldsymbol{\rho}_L(\boldsymbol{v}_0) = \bar{\boldsymbol{v}}$. On the other hand, by virtue of (31), we have $\boldsymbol{\pi}_L(\boldsymbol{v}_0) = \bar{\boldsymbol{v}}$. Therefore,

$$\forall \boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{\rho}_L(\boldsymbol{x}) = \boldsymbol{\pi}_L(\boldsymbol{x}).$$

The proposition is proven. □

The procedure for approximate computation of a pseudoprojection on a linear manifold is presented in the form of Algorithm 1. Let us briefly comment on the steps of this algorithm.

Algorithm 1 Computing of pseudoprojection $\boldsymbol{\rho}_{\mathcal{J}}(\boldsymbol{x})$

Require: $H_i = \{\boldsymbol{x} \in \mathbb{R}^n | \langle \boldsymbol{a}_i, \boldsymbol{x} \rangle = b_i\}$; $\mathcal{J} \subseteq \{1, \ldots, m\}$; $\mathcal{J} \neq \emptyset$; $\bigcap\limits_{i \in \mathcal{J}} H_i \neq \emptyset$

```
1:  function ρ_J(x)
2:      k := 0
3:      x_0 := x
4:      repeat
5:          Σ := 0
6:          for i ∈ J do
7:              Σ := Σ + (⟨a_i, x_k⟩ − b_i) a_i / ‖a_i‖^2
8:          end for
9:          x_(k+1) := x_k − Σ / |J|
10:         ξ_max := 0                                  ▷ Maximum residual
11:         for i ∈ J do
12:             ξ_i := ‖ ⟨a_i, x_(k+1)⟩ − b_i ‖
13:             if ξ_i > ξ_max then
14:                 ξ_max := ξ_i
15:             end if
16:         end for
17:         k := k + 1
18:     until ξ_max < ε_ξ                              ▷ Small parameter ε_ξ > 0
19:     return x_k
20: end function
```

Step 2 sets the iteration counter k to zero. Step 3 sets the initial approximation $\boldsymbol{x}_0$. Step 4 begins the iterative loop of calculating the pseudoprojection. Steps 5–8 calculate the sum from the right side of equation (24) with the current approximation $\boldsymbol{x}_k$. Step 9 finds the next approximation $\boldsymbol{x}_{k+1}$. Steps 10–16 calculate the maximum residual ξ for the next approximation with respect to all hyperplanes H_i involved in the computation. Step 17 increases the iteration counter by one. Step 19 returns the new approximation $\boldsymbol{x}_k$ as a result.

Algorithm 2 Calculation of movement vector $\bar{\boldsymbol{d}} = \boldsymbol{D}(\boldsymbol{u})$

Require: $H_i = \{\boldsymbol{x} \in \mathbb{R}^n | \langle \boldsymbol{a}_i, \boldsymbol{x} \rangle = b_i\}$; $\boldsymbol{u} \in \Gamma(M)$

1: **function** $\boldsymbol{D}(\boldsymbol{u})$
2: $\mathcal{U} := \emptyset$ ▷ $\mathcal{U}$ – set of indices of hyperplanes H_i passing through point $\boldsymbol{u}$
3: **for** $i = 1 \ldots m$ **do**
4: **if** $\langle \boldsymbol{a}_i, \boldsymbol{u} \rangle = b_i$ **then**
5: $\mathcal{U} := \mathcal{U} \cup \{i\}$
6: **end if**
7: **end for**
8: $\bar{\boldsymbol{d}} := \boldsymbol{0}$
9: $f := -\infty$ ▷ f – value of objective function $f(\boldsymbol{x}) = \langle \boldsymbol{c}, \boldsymbol{x} \rangle$
10: $\boldsymbol{e_c} := \boldsymbol{c}/\|\boldsymbol{c}\|$
11: $\boldsymbol{v} := \boldsymbol{u} + \delta \boldsymbol{e_c}$ ▷ Large parameter $\delta > 0$
12: **for** $\mathcal{J} \in \mathcal{P}(\mathcal{U}) \backslash \emptyset$ **do** ▷ $\mathcal{P}(\mathcal{U})$ – set of all subsets of the set $\mathcal{U}$
13: $\boldsymbol{w} := \boldsymbol{\rho}_{\mathcal{J}}(\boldsymbol{v})$
14: $\boldsymbol{d} := \boldsymbol{w} - \boldsymbol{u}$
15: $\boldsymbol{e_d} := \boldsymbol{d}/\|\boldsymbol{d}\|$
16: **if** $(\boldsymbol{u} + \tau \boldsymbol{e_d}) \in M$ **then** ▷ Small parameter $\tau > 0$
17: **if** $\langle \boldsymbol{c}, \boldsymbol{u} + \tau \boldsymbol{e_d} \rangle > f$ **then**
18: $f := \langle \boldsymbol{c}, \boldsymbol{u} + \tau \boldsymbol{e_d} \rangle$
19: $\bar{\boldsymbol{d}} := \boldsymbol{d}$
20: **end if**
21: **end if**
22: **end for**
23: **return** $\bar{\boldsymbol{d}}$
24: **end function**

3. AlFaMove – Along Faces Movement Algorithm

In this section, we describe the new AlFaMove (Along Faces Movement) algorithm, which is a numerical implementation of the surface movement method [11]. The AlFaMove algorithm builds a path on the surface of the feasible polytope from an arbitrary boundary point $\boldsymbol{u}_0 \in M \cap \Gamma(\hat{M})$ to a point $\bar{\boldsymbol{x}}$ that is a solution to LP problem (1). Moving along the faces of an feasible polytope is performed in the direction of maximizing the value of the objective function. The path built as a result of such movement is called the optimal objective path.

The basis of the AlFaMove algorithm is the procedure $\boldsymbol{D}(\boldsymbol{u})$, which calculates at the boundary point $\boldsymbol{u}$ the movement vector $\bar{\boldsymbol{d}}$ along the face of the feasible polytope M in the direction of maximum increase in the value of the objective function. Procedure $\boldsymbol{D}(\boldsymbol{u})$ is presented in the form of Algorithm 2. Geometrical representation of the operation of this algorithm is shown in Fig. 4. Let us briefly comment on the steps of Algorithm 2. Steps 2–7 construct the set $\mathcal{U}$, which includes the indices of all hyperplanes H_i passing through the point $\boldsymbol{u}$. Step 8 resets the direction vector $\bar{\boldsymbol{d}}$. In Step 9, the infinitesimal number[4] is assigned to the variable f, which stores the value of the objective function. Step 10 calculates the unit vector $\boldsymbol{e_c}$ parallel to the vector $\boldsymbol{c}$. In

[4]In the case of double-precision floating-point format that occupies 64 bits in computer memory, the infinitesimal number is $-1 \cdot 10^{308}$.

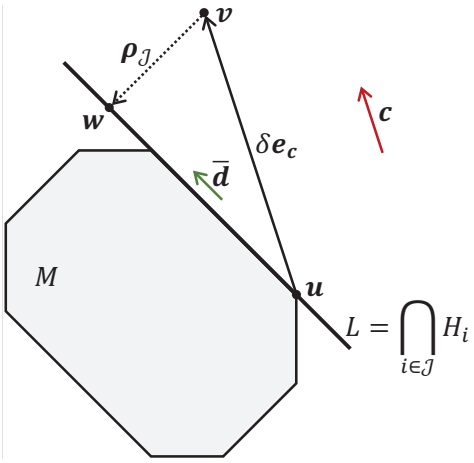

Figure 4. Geometrical representation of Algorithm 2

Step 11, the point $\boldsymbol{v}$ is constructed by adding the vector $\delta\boldsymbol{e_c}$ to the vector $\boldsymbol{u}$ (see Fig. 4). Here, δ is a "large" positive parameter: the greater the δ, the more accurately the direction vector $\bar{\boldsymbol{d}}$ will be calculated. However, when the δ parameter is increased, the time for calculating the pseudoprojection $\boldsymbol{\rho}_{\mathcal{J}}(\boldsymbol{v})$ in Step 13 will also increase. The **for** loop in Step 12 iterates through all possible combinations of indices of the hyperplanes passing through the point $\boldsymbol{u}$. Each such combination $\mathcal{J}$ corresponds to the linear manifold $L = \bigcap_{i\in\mathcal{J}} H_i$, which also passes through the point $\boldsymbol{u}$. Step 13 calculates the point $\boldsymbol{w}$ by pseudoprojecting point $\boldsymbol{v}$ onto the linear manifold corresponding to the combination $\mathcal{J}$. Step 14 calculates, for the current linear manifold, the vector $\boldsymbol{d}$, which determines the direction of maximum increase in the values of the objective function. Step 15 calculates the unit vector $\boldsymbol{e_d}$ parallel to the vector $\boldsymbol{d}$. Step 16 checks that the small movement from point $\boldsymbol{u}$ in the direction $\boldsymbol{d}$ does not exceed the boundaries of the feasible polytope. Step 17, in turn, checks if the value of the objective function at the point $(\boldsymbol{u} + \boldsymbol{e_d})$ is greater than the maximum value obtained in previous iterations of the **for** loop. If so, then the last value is stored as the maximum (Step 18), and the last direction is assigned to vector $\bar{\boldsymbol{d}}$ (Step 19). After all possible combinations have been checked, the vector $\bar{\boldsymbol{d}}$ is returned as a result (Step 23). If none of the combinations passed the check in steps 16–17, the zero vector will be returned as a result. This means that any movement from point $\boldsymbol{u}$ along the surface of the feasible polytope does not lead to an increase in the value of the objective function.

Now, everything is ready to describe the AlFaMove algorithm that solves the LP problem (1). The Algorithm 1 from the paper [11] will serve as a basis for us. The implementation of the AlFaMove algorithm in pseudocode is presented in the form of Algorithm 3. Let us comment on the steps of this algorithm. Step 1 reads the initial approximation $\boldsymbol{u}_0$. This can be an arbitrary boundary point of the recessive polytope $\hat{M}$, satisfying the following condition:

$$\boldsymbol{u}_0 \in M \cap \Gamma(\hat{M}).$$

This condition is checked in Step 2. To obtain a suitable initial approximation, an algorithm can be used that implements the Quest stage of the apex method [17]. Step 3 calculates the initial movement vector $\boldsymbol{d}_0$. To do this, the function $\boldsymbol{D}(\cdot)$ is used, implemented in the Algorithm 2. It is assumed that $\boldsymbol{d}_0 \neq \boldsymbol{0}$[5]. This condition is controlled in Step 4. Step 5 sets the iteration counter k to zero. Step 6 begins the **repeat/until** loop, which performs movement along the

[5]The equality of the vector $\boldsymbol{d}_0$ to the zero vector means that the point $\boldsymbol{u}_0$ is a solution to LP problem (1).

Algorithm 3 AlFaMove

Require: $\hat{H}_i = \{\boldsymbol{x} \in \mathbb{R}^n | \langle \boldsymbol{a}_i, \boldsymbol{x} \rangle \leqslant b_i\}$; $M = \bigcap_{i=1}^{m} \hat{H}_i$; $\hat{M} = \bigcap_{i \in \mathcal{I}} \hat{H}_i$; $i \in \mathcal{I} \Leftrightarrow \langle \boldsymbol{a}_i, \boldsymbol{c} \rangle > 0$

1: **input** $\boldsymbol{u}_0$
2: **assert** $\boldsymbol{u}_0 \in M \cap \Gamma(\hat{M})$
3: $\boldsymbol{d}_0 := \boldsymbol{D}(\boldsymbol{u}_0)$
4: **assert** $\boldsymbol{d}_0 \neq \boldsymbol{0}$
5: $k := 0$
6: **repeat**
7: $\quad \boldsymbol{u}_{k+1} := \boldsymbol{\mu}(\boldsymbol{u}_k, \boldsymbol{d}_k)$
8: $\quad \boldsymbol{d}_{k+1} := \boldsymbol{D}(\boldsymbol{u}_{k+1})$
9: $\quad k := k + 1$
10: **until** $\boldsymbol{d}_k = \boldsymbol{0}$
11: **output** $\boldsymbol{u}_k$ ▷ Solution to LP problem (1)
12: **stop**

faces of the feasible polytope until the movement vector $\boldsymbol{d}_k$ becomes equal to the zero vector. In this case, the last approximation $\boldsymbol{u}_k$ is a solution to LP problem (1). Step 7 calculates the next approximation $\boldsymbol{u}_{k+1}$ using the vector function $\boldsymbol{\mu}$, the definition of which will be given below. Step 8 calculates the movement vector $\boldsymbol{d}_{k+1}$ for the next approximation $\boldsymbol{u}_{k+1}$. Step 9 increases the iteration counter k by one. If the last movement vector is equal to the zero vector, then the **repeat/until** loop is terminated at Step 10, after that, Step 11 outputs the coordinates of the point $\boldsymbol{u}_k$ as a solution to the LP problem (1). Step 12 terminates the AlFaMove algorithm.

The vector function $\boldsymbol{\mu}(\cdot)$ used in Step 7 of Algorithm 3 is defined as follows. Denote

$$\mathcal{Q} = \{i \in \{1, \ldots, m\} \mid \langle \boldsymbol{a}_i, \boldsymbol{u} \rangle < b_i \wedge \langle \boldsymbol{a}_i, \boldsymbol{d} \rangle > 0\}. \tag{32}$$

Then

$$\boldsymbol{\mu}(\boldsymbol{u}, \boldsymbol{d}) = \arg\min_{i \in \mathcal{Q}} \{\|\boldsymbol{u} - \boldsymbol{x}\| \mid \boldsymbol{x} = \boldsymbol{\gamma}_i(\boldsymbol{u}, \boldsymbol{d})\}. \tag{33}$$

Here, $\boldsymbol{\gamma}_i(\boldsymbol{u}, \boldsymbol{d})$ denotes a vector function that calculates the oblique projection of the point $\boldsymbol{u}$ onto the hyperplane H_i relative to the vector $\boldsymbol{d}$:

$$\boldsymbol{\gamma}_i(\boldsymbol{u}, \boldsymbol{d}) = \boldsymbol{u} - \frac{\langle \boldsymbol{a}_i, \boldsymbol{u} \rangle - b_i}{\langle \boldsymbol{a}_i, \boldsymbol{d} \rangle} \boldsymbol{d}.$$

Figure 5 illustrates the action of the function $\boldsymbol{\mu}$. The figure suggests that the hyperplanes H_4 and H_5 do not satisfy the inequality $\langle \boldsymbol{a}_i, \boldsymbol{u} \rangle < b_i$ in equation (32). Hyperplanes H_3 and H_6 do not satisfy the inequality $\langle \boldsymbol{a}_i, \boldsymbol{d} \rangle > 0$ in equation (32). Thus, $\mathcal{Q} = \{1, 2\}$. Since

$$\|\boldsymbol{u} - \boldsymbol{\gamma}_1(\boldsymbol{u}, \boldsymbol{d})\| < \|\boldsymbol{u} - \boldsymbol{\gamma}_2(\boldsymbol{u}, \boldsymbol{d})\|,$$

then $\boldsymbol{\mu}(\boldsymbol{u}, \boldsymbol{d}) = \boldsymbol{\gamma}_1(\boldsymbol{u}, \boldsymbol{d})$.

The following theorem ensures the convergence of Algorithm 3 to a solution of LP problem (1) in a finite number of iterations.

Theorem 1. (Convergence of AlFaMove algorithm) Let the feasible polytope M of LP problem (1) be a bounded nonempty set. Let $\bar{\boldsymbol{x}}$ be a solution to LP problem (1). Then, the sequence

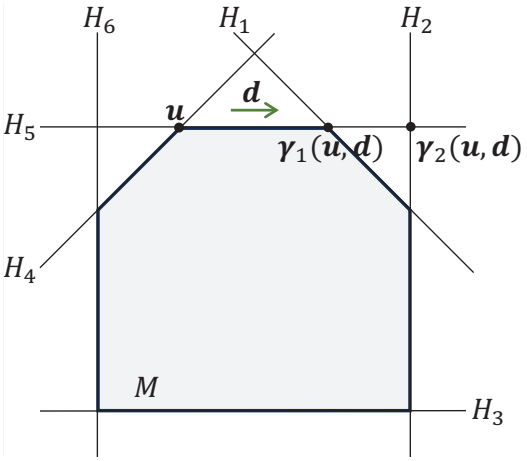

Figure 5. Action of function $\boldsymbol{\mu}$:
$\boldsymbol{\mu}(\boldsymbol{u},\boldsymbol{d}) = \boldsymbol{\gamma}_1(\boldsymbol{u},\boldsymbol{d})$

of approximations $\{\boldsymbol{u}_k\}_{k=1}^{K}$ generated by Algorithm 3, is finite ($K < +\infty$), and, $\langle \boldsymbol{c}, \boldsymbol{u}_K \rangle = \langle \boldsymbol{c}, \bar{\boldsymbol{x}} \rangle$, i.e., $\boldsymbol{u}_K$ is a solution to LP problem (1).

Proof. Denote by $\boldsymbol{d}_{AlFaMove}$ the vector $\boldsymbol{d}_{k+1}$, calculated in step 8 of Algorithm 3. In accordance with steps 13 and 14 of Algorithm 2, the following equation holds:

$$\boldsymbol{d}_{AlFaMove} = \boldsymbol{\rho}_{\mathcal{J}}(\boldsymbol{v}) - \boldsymbol{u}.$$

According to Proposition 2, it follows that

$$\boldsymbol{d}_{AlFaMove} = \boldsymbol{\pi}_{\mathcal{J}}(\boldsymbol{v}) - \boldsymbol{u}, \tag{34}$$

where $\boldsymbol{\pi}_{\mathcal{J}}(\boldsymbol{v})$ denotes the orthogonal projection of the point $\boldsymbol{v}$ onto the linear manifold $L = \bigcap_{i \in \mathcal{J}} H_i$. According to Proposition 1, this means that the vector $\boldsymbol{d}_{AlFaMove}$ uniquely determines the direction of maximum increase in the objective function value of LP problem (1). And this, in turn, means that Algorithm 3 is a numerical implementation of the surface movement method [11], i.e., the approximation sequences of $\left\{\boldsymbol{u}^k_{AlFaMove}\right\}$ and $\left\{\boldsymbol{u}^k_{SMM}\right\}$, generated respectively by Algorithm 3 from this article and Algorithm 1 from [11], coincide. Thus, the convergence of the AlFaMove algorithm directly follows from the convergence of the surface movement algorithm, ensured by Theorem 1 from article [11]. □

4. Parallel Version of AlFaMove Algorithm

The most compute-intensive operation in the AlFaMove algorithm (Algorithm 3) is the operation $\boldsymbol{D}(\cdot)$, which calculates the direction vector at Step 8 in the **repeat/until** loop. When solving large-scale LP problems, it takes more than 90% of the processor time. This is explained by the fact that the vector function $\boldsymbol{D}(\cdot)$, implemented as Algorithm 2, uses, at Step 13, the pseudoprojection operation $\boldsymbol{\rho}_{\mathcal{J}}(\cdot)$[6], which repeatedly applies the mapping $\varphi(\cdot)$ defined by equation (24) to the starting point $\boldsymbol{v}$. This iterative method uses the orthogonal projection of a point onto a hyperplane as an elementary operation and belongs to the class of projection methods. It is known that in the case of large-scale LP problems, the projection method may require

[6]See Definition 3 and Algorithm 1.

significant time costs [6]. In addition, it should be noted that Algorithm 2 at Step 12 iterates through all non-empty subsets of the set $\mathcal{U}$, which includes the indices of hyperplanes passing through the point $\boldsymbol{u}$. For example, if 30 hyperplanes pass through a point, then we will have $2^{30} - 1 = 1\,073\,741\,823$ non-empty subsets. To iterate through such a number of subsets, we will need the power of a supercomputer. Therefore, we have developed a parallel version of the AlFaMove algorithm, presented as Algorithm 4. Parallel Algorithm 4 is based on the BSF par-

Algorithm 4 Parallel version of AlFaMove algorithm

master		***l*th worker** $(l = 0, \ldots, L-1)$	
1:	**input** $n, m, A, \boldsymbol{b}, \boldsymbol{u}_0$	1:	**input** $n, m, A, \boldsymbol{b}, \boldsymbol{c}$
2:	$k := 0$	2:	
3:	**repeat**	3:	**repeat**
4:	**Broadcast** $\boldsymbol{u}_k$	4:	**RecvFromMaster** $\boldsymbol{u}_k$
5:		5:	$\mathcal{U} :=[\,]$
6:		6:	**for** $i = 1 \ldots m$ **do**
7:		7:	**if** $\langle \boldsymbol{a}_i, \boldsymbol{u}_k \rangle = b_i$ **then**
8:		8:	$\mathcal{U} := \mathcal{U} + [i]$
9:		9:	**end if**
10:		10:	**end for**
11:		11:	$K := 2^{\lvert\mathcal{U}\rvert} - 1$
12:		12:	$L :=$ NumberOfWorkers
13:		13:	$\mathcal{L}_{map(l)} :=[lK/L, \ldots, (l+1)K/L - 1]$
14:		14:	$\mathcal{L}_{reduce(l)} := Map(\mathrm{F}_{\boldsymbol{u}_k}, \mathcal{L}_{map(l)})$
15:		15:	$(\boldsymbol{d}_l, f_l) := Reduce(\oplus, \mathcal{L}_{reduce(l)})$
16:	**Gather** $\mathcal{L}_{reduce}$	16:	**SendToMaster** $(\boldsymbol{d}_l, f_l)$
17:	$(\boldsymbol{d}_k, f_k) := Reduce(\oplus, \mathcal{L}_{reduce})$	17:	
18:	**if** $\boldsymbol{d}_k = \boldsymbol{0}$ **then**	18:	
19:	$exit :=$ **true**	19:	
20:	**else**	20:	
21:	$\boldsymbol{u}_{k+1} := \boldsymbol{\mu}(\boldsymbol{u}_k, \boldsymbol{d}_k)$	21:	
22:	$k := k + 1$	22:	
23:	$exit :=$ **false**	23:	
24:	**end if**	24:	
25:	**Broadcast** $exit$	25:	**RecvFromMaster** $exit$
26:	**until** $exit$	26:	**until** $exit$
27:	**output** $\boldsymbol{u}_k, f_k$	27:	
28:	**stop**	28:	**stop**

allel computation model [14] designed for cluster computing systems. The BSF model uses the master–worker parallelization scheme and requires the representation of the algorithm in the form of operations on lists using higher-order functions *Map* and *Reduce*. In Algorithm 4, the higher-order function *Map* takes, as the second parameter, the list $\mathcal{L}_{map} = [1, \ldots, K]$ containing the ordinal numbers of all subsets of the set $\mathcal{U}$, with the exception of the empty set. Here,

$K = 2^{|\mathcal{U}|} - 1$. As the first parameter, *Map* takes the parameterized function

$$\mathrm{F}_{\boldsymbol{u}} : \{1, \ldots, K\} \to \mathbb{R}^n \times \mathbb{R},$$

which is defined as follows:

$$\begin{aligned} &\mathrm{F}_{\boldsymbol{u}}(j) = (\boldsymbol{d}_j, f_j); \\ &\boldsymbol{d}_j = \begin{cases} \boldsymbol{e_d}, & \text{if } (\boldsymbol{u} + \tau \boldsymbol{e_d}) \in M \wedge \langle \boldsymbol{c}, \boldsymbol{w} \rangle > \langle \boldsymbol{c}, \boldsymbol{u} \rangle; \\ \boldsymbol{0}, & \text{if } (\boldsymbol{u} + \tau \boldsymbol{e_d}) \notin M \vee \langle \boldsymbol{c}, \boldsymbol{w} \rangle \leqslant \langle \boldsymbol{c}, \boldsymbol{u} \rangle; \end{cases} \\ &f_j = \begin{cases} \langle \boldsymbol{c}, \boldsymbol{u} + \boldsymbol{e_d} \rangle, & \text{if } (\boldsymbol{u} + \tau \boldsymbol{e_d}) \in M \wedge \langle \boldsymbol{c}, \boldsymbol{w} \rangle > \langle \boldsymbol{c}, \boldsymbol{u} \rangle; \\ -\infty, & \text{if } (\boldsymbol{u} + \tau \boldsymbol{e_d}) \notin M \vee \langle \boldsymbol{c}, \boldsymbol{w} \rangle \leqslant \langle \boldsymbol{c}, \boldsymbol{u} \rangle, \end{cases} \end{aligned} \tag{35}$$

where

$$\boldsymbol{w} = \boldsymbol{\rho}_{\sigma(j)}(\boldsymbol{u} + \delta \boldsymbol{c}/\|\boldsymbol{c}\|), \tag{36}$$

and

$$\boldsymbol{e_d} = \frac{\boldsymbol{w} - \boldsymbol{u}}{\|\boldsymbol{w} - \boldsymbol{u}\|}.$$

The semantics of the function $\mathrm{F}_{\boldsymbol{u}}(\cdot)$ is uniquely determined by Algorithm 2. The function $\sigma(\cdot)$ used in equation (36) maps the natural number $j \in \{1, \ldots, K\}$ to the jth subset of the set that includes all the elements of the list $\mathcal{U}$. To do this, the number j is converted to a binary representation consisting of $|\mathcal{U}|$ bits. Each bit corresponds to the hyperplane index from the list $\mathcal{U}$ in natural order. If the bit contains 1, then the corresponding index is included in the subset $\sigma(j)$. If the bit contains 0, then the corresponding index is not included. For example, let the hyperplanes H_2, H_4, H_7, H_9 pass through the point $\boldsymbol{u}$. In this case, $\mathcal{U} = [2, 4, 7, 9]$ and $K = 2^4 - 1 = 15$, i.e., 15 different non-empty subsets can be formed from the set of elements of the list $\mathcal{U}$. For instance, let us find the fifth subset. The function $\sigma(\cdot)$ converts the number 5 into the binary representation of 4 bits 0101 and returns the subset $\{4, 9\}$ as a result. In such a way, the higher-order function $Map\,(\mathrm{F}_{\boldsymbol{u}}, \mathcal{L}_{map})$ converts the list $\mathcal{L}_{map}$ of ordinal numbers of subsets into the list of pairs $(\boldsymbol{d}_j, f_j)$:

$$Map\,(\mathrm{F}_{\boldsymbol{u}}, \mathcal{L}_{map}) = [\mathrm{F}_{\boldsymbol{u}}(1), \ldots, \mathrm{F}_{\boldsymbol{u}}(K)] = [(\boldsymbol{d}_1, f_1), \ldots, (\boldsymbol{d}_K, f_K)]\,.$$

Here, $\boldsymbol{d}_j$ $(j = 1, \ldots, K)$ is the movement unit vector, and f_j is the value of the objective function, which is reached at the point $\boldsymbol{u} + \boldsymbol{d}_j$.

Denote by $\mathcal{L}_{reduce}$ the list of pairs generated by the higher-order function Map:

$$\mathcal{L}_{reduce} = Map\,(\mathrm{F}_{\boldsymbol{u}}, \mathcal{L}_{map}) = [(\boldsymbol{d}_1, f_1), \ldots, (\boldsymbol{d}_K, f_K)]\,.$$

Define the binary associative operation

$$\oplus : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n \times \mathbb{R},$$

which is the first parameter of the higher-order function *Reduce*:

$$\left(\boldsymbol{d}', f'\right) \oplus \left(\boldsymbol{d}'', f''\right) = \begin{cases} \left(\boldsymbol{d}', f'\right), & \text{if } f' \geqslant f''; \\ \left(\boldsymbol{d}'', f''\right), & \text{if } f' < f''. \end{cases} \tag{37}$$

Higher-order function *Reduce* reduces the list $\mathcal{L}_{reduce}$ to a single pair by sequentially applying the operation $\oplus$ to all elements of the list:

$$Reduce\,(\oplus, \mathcal{L}_{reduce}) = (\boldsymbol{d}_1, f_1) \oplus \ldots \oplus (\boldsymbol{d}_K, f_K) = (\boldsymbol{d}_{j'}, f_{j'}),$$

where, according to (37)

$$j' = \arg \max_{1 \leqslant j \leqslant K} f_j.$$

Parallel Algorithm 4 uses the master–worker approach and includes $L + 1$ process: one process is the master and L processes are the workers. The master process performs general computing management, distributes work between worker processes, receives results from them and generates the final result. For simplicity, we assume that the subset number K is a multiple of the number of workers L. In Step 1, the master reads the initial data of the LP problem and the coordinates of the starting point $\boldsymbol{u}_0$. In step 2, the master sets the iteration counter k to zero. Steps 3–26 implement the main loop **repeat/until** calculating the solution to LP problem (1). In Step 4, the master broadcasts the current approximation $\boldsymbol{u}_k$ to all workers. In Step 16, the master receives from the workers the partial results, which are reduced to the single pair $(\boldsymbol{d}_k, f_k)$ in Step 17. If the condition $\boldsymbol{d}_k = \boldsymbol{0}$ is met in Step 18, then a solution is found (we assume that $\boldsymbol{d}_0 \neq \boldsymbol{0}$). In this case, the master assigns the value **true** to the Boolean variable *exit* in Step 19. If $\boldsymbol{d}_k \neq \boldsymbol{0}$, then the master calculates the next approximation $\boldsymbol{u}_{k+1}$ in Step 21, increases the iteration counter k by one in Step 22, and assigns the value **false** to the Boolean variable *exit* in Step 23. In Step 25, the master broadcasts the value of the Boolean variable *exit* to all workers. If the Boolean variable *exit* takes the value **true**, then the **repeat/until** loop ends in Step 26. In Step 27, the master outputs the last approximation $\boldsymbol{u}_k$ as a result, and the quantity f_k as the optimal value of the objective function. Step 28 terminates the master process.

All workers execute the same code, but on different data. In Step 1, the lth worker $(l = 1, \ldots, L)$ reads the initial data of the LP problem. The **repeat/until** loop of the worker (steps 3–26) corresponds to the **repeat/until** loop of the master. In Step 4, the worker receives the current approximation $\boldsymbol{u}_k$ from the master. After that, the worker forms its own sublist $\mathcal{L}_{map(l)}$ of the subset ordinal numbers to be processed (steps 5–13). The sublists of different workers do not overlap:

$$l' \neq l'' \Leftrightarrow \mathcal{L}_{map(l')} \neq \mathcal{L}_{map(l'')}, \tag{38}$$

and their concatenation gives a complete list:

$$\mathcal{L}_{map} = \mathcal{L}_{map(0)} +\!\!+ \ldots +\!\!+ \mathcal{L}_{map(L-1)}. \tag{39}$$

In Step 14, the worker calls the higher-order function *Map*, which forms the sublist of pairs $\mathcal{L}_{reduce(l)}$, applying the parameterized function $\mathrm{F}_{\boldsymbol{u}_k}$, defined by the equations (35), to all elements of the sublist $\mathcal{L}_{map(l)}$. In Step 15, the higher-order function *Reduce* transforms this list into the single pair $(\boldsymbol{d}_l, f_l)$ by sequentially applying the binary operation $\oplus$, defined by the equation (37), to all elements of the sublist $\mathcal{L}_{reduce(l)}$. The result is sent to the master in Step 16. In Step 25, the worker receives the value of the Boolean variable *exit* from the master. If this variable takes the value **true**, then the worker process is terminated. Otherwise, the **repeat/until** loop continues to run. The exchange operators **Broadcast**, **Gather**, **RecvFromMaster** and **SendToMaster** perform implicit synchronization of the master process and worker processes.

5. Computational Experiments

We implemented the parallel version of the AlFaMove algorithm in C++ using the BSF-skeleton [15], which is based on the BSF parallel computation model [14]. The BSF-skeleton encapsulates all aspects related to parallelizing a program based on the MPI library. The source codes of the parallel implementation of the AlFaMove algorithm are freely available in the GitHub repository at `https://github.com/leonid-sokolinsky/AlFaMove`. The developed program has been tested on a large number of LP problems from various sources. All these problems in MTX format [1] are available at `https://github.com/leonid-sokolinsky/Set-of-LP-Problems`. As tests, we also used synthetic problems obtained using the random problem generator LP FRaGenLP [16]. These problems are available at `https://github.com/leonid-sokolinsky/Set-of-LP-Problems/tree/main/Rnd-LP`. We were unable to test the AlFaMove implementation on problems from the Netlib-LP repository [5], since, in all these problems, the number of hyperplanes passing through the starting point $\boldsymbol{u}_0$ exceeded the number 30, which corresponds to the number of possible combinations equal to 1 073 741 824. The C++ compilers available to us do not accept arrays of such sizes.

Using the developed program, we evaluated the scalability of the AlFaMove algorithm. In these experiments, we used the parameterized LP problem called "cut-off vertex hypercube", for which the space dimension n is a parameter. The constraints of this problem contain the following $2n+1$ inequalities:

$$\begin{cases} x_1 & & & & & \leqslant & 200 \\ & x_2 & & & & \leqslant & 200 \\ & & & & & \vdots & \\ & & & x_n & & \leqslant & 200 \\ x_1 + & x_2 & \cdots & + & x_n & \leqslant & 200(n-1)+100 \\ x_1 \geqslant 0, & x_2 \geqslant 0, & \cdots & , x_n \geqslant 0. & & & \end{cases} \tag{40}$$

The gradient of the objective function is given by the vector

$$\boldsymbol{c} = (1, 2, \ldots, n)\,. \tag{41}$$

It is necessary to find the maximum of the objective function. The problem has the unique solution at the point $(100, 200, \ldots, 200)$ with the maximum value of the objective function equal to $100(n^2+n-1)$. For an arbitrary n, this problem can be obtained in MTX format using the FRaGenLP generator, if the number of random inequalities is set to 0. With various n, these LP problems are available at `https://github.com/leonid-sokolinsky/Set-of-LP-Problems/tree/main/Rnd-LP` under the names `lp_rnd<n>-0`, where the dimension of the space is specified as `<n>`.

Computational experiments were carried out on the supercomputer "Lomonosov-2" [19], whose specifications are shown in Tab. 1.
All computations were performed with double precision, at which a floating-point number occupies 64 bits in computer memory.

In the first series of experiments, the dependence of the speedup and parallel efficiency of the AlFaMove algorithm on the number of processor nodes for the cut-off vertex hypercube problem was investigated. The results of these experiments are shown in Fig. 6. The speedup $\alpha(L)$ was defined as the ratio of the time $T(1)$ of solving a problem in the configuration with

Table 1. Specifications of "Lomonosov-2" computing cluster

Parameter	Value
Number of processor nodes	1487
Processor	Intel Haswell-EP E5-2697v3, 2.6 GHz, 14 cores
Memory per node	64 GB
Main network	InfiniBand FDR
Control network	Gigabit Ethernet
Operating system	Linux CentOS 7

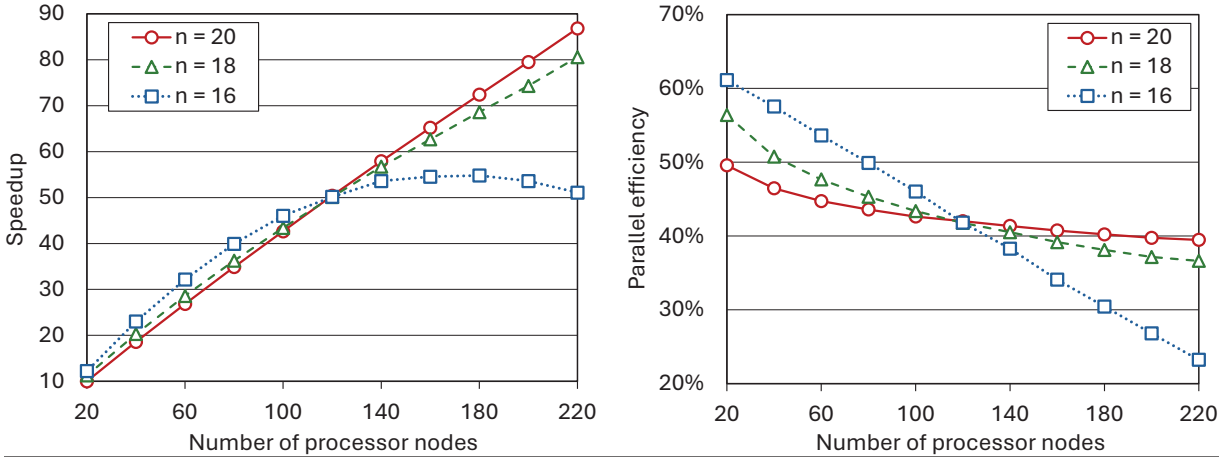

Figure 6. Speedup and parallel efficiency of the AlFaMove algorithm, n – number of variables in LP problem (40)

a master node and a single worker node to the time $T(L)$ of solving the same problem in the configuration with a master node and L worker nodes:

$$\alpha(L) = \frac{T(1)}{T(L)}.$$

Parallel efficiency $\beta(L)$ was calculated using the equation

$$\beta(L) = \frac{T(1)}{L \cdot T(L)}.$$

Computations were performed for the dimensions 16, 18 and 20. The number of constraints was 33, 37 and 41 respectively. In all cases, the vertex of the feasible polytope with the following coordinates was chosen as the initial point:

$$x_1 = 0, \quad \ldots \quad x_{n/2} = 0, \quad x_{n/2+1} = 200, \quad \ldots \quad x_n = 200. \tag{42}$$

The experiments demonstrated good scalability of the AlFaMove algorithm on the cut-off vertex hypercube problem, starting from the dimension $n = 18$. In this case, the algorithm demonstrated speedup close to linear. At smaller dimensions, the cost of exchanges and latency begin to dominate the computational costs, which leads to a significant decrease in the algorithm scalability boundary[7]. For $n = 16$, this boundary was equal to 180 nodes. Experiments also shown

[7]The scalability boundary refers to the maximum number of processor nodes, up to which the speedup increases.

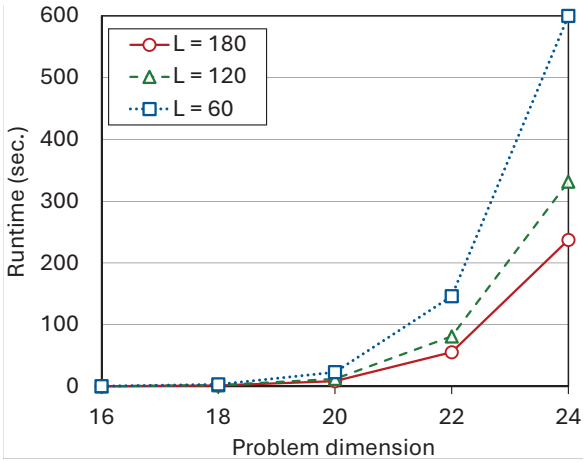

Figure 7. The dependence of AlFaMove runtime on the problem dimension on various multiprocessor configurations (L – number of processor nodes)

that with an increase in the problem dimension, the parallel efficiency on a small number of processor nodes (less than 120) decreases. However, with a larger number of processor nodes, the opposite trend is observed. So, for the dimension $n = 16$, the parallel efficiency was 61% on 20 processor nodes, after which it decreased to 23% on 220 nodes. At the same time, for $n = 20$, the parallel efficiency was equal to 50% and 40%, respectively.

In the next series of experiments, the dependence of runtime on the dimension of the cut-off vertex hypercube problem was investigated for various multiprocessor configurations with the number of processor nodes $L = 60$, $L = 120$ and $L = 180$. The results of these experiments are shown in Fig. 7. The dimension ranged from $n = 16$ to $n = 24$ in increments of 2. For the dimension $n = 24$, each of the lists $\mathcal{L}_{map}$ and $\mathcal{L}_{reduce}$ included 16 777 215 elements. This is the maximum size allowed by the compiler used. The vertex of the feasible polytope with coordinates (42) was always chosen as the initial point. In all the studied configurations, the experiments showed an exponential increase in the runtime with an increase in the problem dimension. However, configurations with a large number of processor nodes demonstrated considerably shorter running time of the AlFaMove algorithm.

In the third series of experiments, we investigated the behavior of the AlFaMove algorithm on the Klee–Minty cube. The feasible region of this problem is a hypercube with perturbed corners defined by the following inequalities:

$$\begin{cases} x_1 & & & & & & & \leqslant & 5 \\ 4x_1 & + & x_2 & & & & & \leqslant & 25 \\ 8x_1 & + & 4x_2 & + & x_3 & & & \leqslant & 125 \\ & & & & & & & \vdots & \\ 2^n x_1 & + & 2^{n-1}x_2 & \cdots & + & 4x_{n-1} & + & x_n \leqslant & 5^n \\ x_1 \geqslant 0, & & x_2 \geqslant 0, & \cdots & & , x_n \geqslant 0. & & & \end{cases}$$

The gradient of the objective function is given by the vector

$$\boldsymbol{c} = \left(2^{n-1}, 2^{n-2}, \ldots, 2, 1\right).$$

It is necessary to find the maximum of the objective function. The problem has the unique solution at the point $(0, \ldots, 0, 5^n)$ with the maximum value of the objective function equal

to 5^n. In article [7], Victor Klee and George Minty showed that the classical simplex method, starting at $\boldsymbol{x} = \mathbf{0}$, goes through all 2^n hypercube vertices performing $2^n - 1$ iterations in solving this problem. It is known that many optimization algorithms for linear programming exhibit poor performance when applied to the Klee–Minty cube. We applied the AlFaMove algorithm to Klee–Minty cubes of dimension from 5 to 9. The experimental results presented in Tab. 2 show that the AlFaMove algorithm found a solution in $2n - 1$ iterations in all cases, while the simplex method performed $2^n - 1$ iterations.

Table 2. Experiments with Klee–Minty cubes

Dimension n	AlFaMove				Simplex
	Scalability boundary	Time (sec.)	Relative error δ	Iteration number	Iteration number
5	10	0.2	$0.9 \cdot 10^{-12}$	9	31
6	15	2	$0.2 \cdot 10^{-12}$	11	63
7	20	13	$0.8 \cdot 10^{-11}$	13	127
8	25	126	$0.8 \cdot 10^{-11}$	15	255
9	30	1445	$0.2 \cdot 10^{-10}$	17	511

The relative error was calculated by the equation

$$\delta = \left| \frac{f_{exact} - f_{approx}}{f_{exact}} \right|,$$

where f_{exact} is the exact maximum value of the objective function, f_{approx} is the value calculated by the AlFaMove algorithm. The iterations of the simplex method were calculated using the online calculator available at `https://www.pmcalculators.com/simplex-method-calculator`. Experiments also showed that in the case of Klee–Minty cubes, the scalability boundary of the AlFaMove algorithm increased linearly with increasing the problem dimension. At the same time, an exponential increase in runtime was observed.

Conclusion

The article presents the AlFaMove algorithm, which is a numerical implementation of the surface movement method for linear programming. The key feature of this method is to find out the optimal path along the surface of the feasible polytope from the initial point to a solution of a linear programming problem. The optimal path is understood as a path along the surface of the feasible region in the direction of maximizing the values of the objective function. The scientific significance of the proposed algorithm lies in the fact that it opens up the possibility of using feed forward artificial neural networks to solve non-stationary multidimensional linear programming problems in real time. The theoretical basis of the AlFaMove algorithm is the operation of constructing the pseudoprojection onto linear manifolds that form the feasible polytope flat sides of different dimensions.

Pseudoprojection is implemented on the basis of the Fejér process and is a generalization of the concepts of orthogonal projection on a linear manifold and metric projection on a convex

set. In the case of a hyperplane, the pseudoprojection turns into the orthogonal projection. It is proved that the hyperplane path constructed by the gradient of the objective function and the orthogonal projection is optimal. The projection-type algorithm is presented for constructing a pseudoprojection onto linear manifold formed by the hyperplane intersections. It is proven that the pseudoprojection point coincides with the orthogonal projection point in this case. A formalized description of the AlFaMove algorithm, which builds the optimal path on the surface of the feasible polytope, is presented. The AlFaMove algorithm is based on the procedure for calculating the vector of movement along the face of the feasible polytope from the current approximation in the direction of maximizing the values of the objective function. A formalized description of this procedure is outlined.

Projection-type algorithms are characterized by a low rate of convergence, depending on the angles between the hyperplanes forming the linear manifold. It is also noted that the calculation of the movement vector is a combinatorial-type enumeration problem with high space and time complexity. A parallel version of the AlFaMove algorithm designed for cluster computing systems is presented. The parallel version is implemented in C++ using the BSF-skeleton based on the BSF parallel computation model. Computational experiments were conducted on a cluster computing system to evaluate the scalability of the AlFaMove algorithm. The experiments showed that a linear programming problem with 24 variables and 49 constraints demonstrates a speedup close to linear on 320 processor nodes of the cluster. Problems of a larger dimension led to a compiler error caused by exceeding the maximum acceptable size of arrays. The experiments with the Klee–Minty cube shown that the scalability boundary of the AlFaMove algorithm also increases linearly with increasing the problem dimension.

As directions for further research, we outline the following. We plan to design a new, more efficient method for constructing a path on the surface of a feasible polytope, leading to a solution of a linear programming problem. The main idea is to decrease the number of enumerating combinations of hyperplanes when determining the direction of movement. This can be achieved by restricting the paths of movement only to the edges of the polytope (segments of linear manifolds of dimension one). The problem of space complexity can be solved by using stochastic methods of choosing the movement direction.

Acknowledgements

The research was supported by the Russian Science Foundation (project No. 23-21-00356) and carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University.

References

1. Boisvert, R.F., Pozo, R., Remington, K.A.: The Matrix Market Exchange Formats: Initial Design. Tech. rep., NISTIR 5935. National Institute of Standards and Technology, Gaithersburg, MD (1996), `https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir5935.pdf`, accessed: 2024-08-26

2. Branke, J.: Optimization in Dynamic Environments. In: Evolutionary Optimization in Dynamic Environments. Genetic Algorithms and Evolutionary Computation, vol. 3, pp. 13–29. Springer, Boston, MA (2002). https://doi.org/10.1007/978-1-4615-0911-0_2

3. Dantzig, G.B.: Linear programming and extensions. Princeton university press, Princeton, N.J. (1998)

4. Fathi, M., Khakifirooz, M., Pardalos, P.M. (eds.): Optimization in Large Scale Problems: Industry 4.0 and Society 5.0 Applications. Springer, Cham, Switzerland (2019). https://doi.org/10.1007/978-3-030-28565-4

5. Gay, D.M.: Electronic mail distribution of linear programming test problems. Mathematical Programming Society COAL Bulletin 13, 10–12 (1985)

6. Gould, N.I.: How good are projection methods for convex feasibility problems? Computational Optimization and Applications 40(1), 1–12 (2008). https://doi.org/10.1007/S10589-007-9073-5

7. Klee, V., Minty, G.J.: How good is the simplex algorithm? In: Shisha, O. (ed.) Inequalities - III. Proceedings of the Third Symposium on Inequalities Held at the University of California, Los Angeles, Sept. 1-9, 1969. pp. 159–175. Academic Press, New York, NY, USA (1972)

8. Kopanos, G.M., Puigjaner, L.: Solving Large-Scale Production Scheduling and Planning in the Process Industries. Springer, Cham, Switzerland (2019). https://doi.org/10.1007/978-3-030-01183-3

9. Maltsev, A.: The basics of linear algebra. Science. The main editorial office of the phys-math literature, Moskow (1970), (in Russian)

10. Mamalis, B., Pantziou, G.: Advances in the Parallelization of the Simplex Method. In: Zaroliagis, C., Pantziou, G., Kontogiannis, S. (eds.) Algorithms, Probability, Networks, and Games. Lecture Notes in Computer Science, vol. 9295, pp. 281–307. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24024-4_17

11. Olkhovsky, N.A., Sokolinsky, L.B.: Surface Movement Method for Linear Programming. Lobachevskii Journal of Mathematics 45(10), (in print) (2024)

12. Schlenkrich, M., Parragh, S.N.: Solving large scale industrial production scheduling problems with complex constraints: an overview of the state-of-the-art. In: Longo, F., Affenzeller, M., Padovano, A., Shen, W. (eds.) 4th International Conference on Industry 4.0 and Smart Manufacturing. Procedia Computer Science. vol. 217, pp. 1028–1037. Elsevier (2023). https://doi.org/10.1016/J.PROCS.2022.12.301

13. Sokolinskaya, I.M., Sokolinsky, L.B.: On the Solution of Linear Programming Problems in the Age of Big Data. In: Sokolinsky, L., Zymbler, M. (eds.) Parallel Computational Technologies. PCT 2017. Communications in Computer and Information Science, vol. 753. pp. 86–100. Springer, Cham, Switzerland (2017). https://doi.org/10.1007/978-3-319-67035-5_7

14. Sokolinsky, L.B.: BSF: A parallel computation model for scalability estimation of iterative numerical algorithms on cluster computing systems. Journal of Parallel and Distributed Computing 149, 193–206 (2021). https://doi.org/10.1016/j.jpdc.2020.12.009

15. Sokolinsky, L.B.: BSF-skeleton: A Template for Parallelization of Iterative Numerical Algorithms on Cluster Computing Systems. MethodsX 8, Article number 101437 (2021). https://doi.org/10.1016/j.mex.2021.101437

16. Sokolinsky, L.B., Sokolinskaya, I.M.: FRaGenLP: A Generator of Random Linear Programming Problems for Cluster Computing Systems. In: Sokolinsky, L., Zymbler, M. (eds.) Parallel Computational Technologies. PCT 2021. Communications in Computer and Information Science, vol. 1437. pp. 164–177. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81691-9_12

17. Sokolinsky, L.B., Sokolinskaya, I.M.: Apex Method: A New Scalable Iterative Method for Linear Programming. Mathematics 11(7), 1–28 (2023). https://doi.org/10.3390/MATH11071654

18. Vasin, V.V., Eremin, I.I.: Operators and Iterative Processes of Fejér Type. Theory and Applications. Inverse and Ill-Posed Problems Series, Walter de Gruyter, Berlin, New York (2009). https://doi.org/10.1515/9783110218190

19. Voevodin, V., Antonov, A., Nkitenko, D., Shvets, P., Sobolev, S., Sidorov, I., Stefanov, K., Voevodin, V., Zhumatiy, S.: Supercomputer Lomonosov-2: Large Scale, Deep Monitoring and Fine Analytics for the User Community. Supercomputing Frontiers and Innovations 6(2), 4–11 (2019). https://doi.org/10.14529/jsfi190201

20. Zorkaltsev, V., Mokryi, I.: Interior point algorithms in linear optimization. Journal of applied and industrial mathematics 12(1), 191–199 (2018). https://doi.org/10.1134/S1990478918010179