Report 2

Team information (B23-ISE-02):

- Team member 1 (leader): Kirill Efimovich (k.efimovich@innopolis.university), grade 5/5
- Team member 2: Arsen Galiev (a.galiev@innopolis.university), grade: 5/5
- Team member 3: Asqar Arslanov (a.arslanov@innopolis.university), grade: 5/5
- Team member 4: Ilya-Linh Nguen (i.nguen@innopolis.university), grade: 5/5

Link to the product:

- `https://github.com/quintet-sdr/optimization-pt2`

Programming language:

- Rust
- To launch the code: `$ cargo run`

Linear programming problem:

- Maximization
- Approximation accuracy $\epsilon = 0.005$

Tests:

(1) **Input:**

- Objective function: $F(x_1, x_2) = 10x_1 + 20x_2$
- Constraints: $\begin{cases} -x_1 + 2x_2 \leq 15 \\ x_1 + x_2 \leq 12 \\ 5x_1 + 3x_2 \leq 45 \\ x_1, x_2 \geq 0 \end{cases}$

Output:

- Solver state: solved
- Decision variables: $\{3, 9\}$
- Maximum value: 210

(2) **Input:**

- Objective function: $F(x_1, x_2, x_3) = 9x_1 + 10x_2 + 16x_3$
- Constraints: $\begin{cases} 18x_1 + 15x_2 + 12x_3 \leq 360 \\ 6x_1 + 4x_2 + 8x_3 \leq 192 \\ 5x_1 + 3x_2 + 3x_3 \leq 180 \\ x_1, x_2, x_3 \geq 0 \end{cases}$

Output:

- Solver state: solved
- Decision variables: $\{0, 8, 20\}$
- Maximum value: 400

(3) **Input:**

- Objective function: $F(x_1, x_2, x_3) = 2x_1 - 2x_2 + 6x_3$
- Constraints: $\begin{cases} 2x_1 + x_2 - 2x_3 \leq 24 \\ x_1 + 2x_2 + 4x_3 \leq 23 \\ x_1 - x_2 + 2x_3 \leq 10 \\ x_1, x_2, x_3 \geq 0 \end{cases}$

Output:

- Solver state: solved
- Decision variables: $\{0, 0.75, 5.375\}$
- Maximum value: 30.75

(4) **Input:**

- Objective function: $F(x_1, x_2) = 2x_1 + x_2$
- Constraints: $\begin{cases} x_1 - x_2 \leq 10 \\ -2x_1 \leq 40 \\ x_1, x_2 \geq 0 \end{cases}$

Output:

- Solver state: unbounded

(5) **Input:**

- Objective function: $F(x_1, x_2) = 3x_1 + 2x_2$
- Constraints: $\begin{cases} x_1 - x_2 \leq 2 \\ -2x_1 + x_2 \leq -1 \\ x_1, x_2 \geq 0 \end{cases}$

Output:

- Solver state: unbounded

Code:

crates/pt2-cli/src/main.rs

```
use crate::tests::Lpp;

mod tests;

fn main() {
    const EPS: usize = 2;
    const ALPHA_1: f64 = 0.5;
    const ALPHA_2: f64 = 0.9;

    for generate_test in tests::generators() {
        for alpha in [ALPHA_1, ALPHA_2] {
            let Lpp {
                objective_function,
                constraints,
                initial_point,
            } = generate_test();

            let iterations = match pt2_core::interior_point(
                objective_function,
                constraints,
                initial_point,
                EPS,
                alpha,
            ) {
                Ok(it) => it,
                Err(err) => {
                    println!("{err}");
                    continue;
                }
            };

            let last = iterations.last().unwrap();

            let result = match last {
                Ok(it) => it,
                Err(err) => {
                    println!("{err}");
                    continue;
                }
            };

            println!("alpha: {alpha:.EPS$}");
            println!("max: {:.EPS$}", result.max);
            println!("x:{:.EPS$}", result.decision_variables.transpose());
        }
    }
}
```

crates/pt2-cli/src/tests.rs

```
use pt2_core::{Constraints, Sign};

pub struct Lpp<’a> {
    pub objective_function: Vec<f64>,
    pub constraints: Constraints<’a>,
    pub initial_point: Vec<f64>,
}

pub fn generators<’a>() -> &’a [fn() -> Lpp<’a>] {
    &[lab_6_problem_1, lab_6_problem_2]
}

fn lab_6_problem_1<’a>() -> Lpp<’a> {
    Lpp {
        objective_function: vec![1., 1.],
        constraints: &[(&[2., 4.], Sign::Le, 16.), (&[1., 3.], Sign::Ge, 9.)],
        initial_point: vec![0.5, 3.5, 1., 2.],
    }
}

fn lab_6_problem_2<’a>() -> Lpp<’a> {
    Lpp {
        objective_function: vec![9., 10., 16.],
        constraints: &[
            (&[18., 15., 12.], Sign::Le, 360.),
            (&[6., 4., 8.], Sign::Le, 192.),
            (&[5., 3., 3.], Sign::Le, 180.),
        ],
        initial_point: vec![1., 1., 1., 315., 174., 169.],
    }
}
```

crates/pt2-core/src/lib.rs

```
use na::{DMatrix, DVector};

pub use crate::interfaces::{Constraints, Sign};
use crate::interfaces::{InteriorPoint, NotApplicableError};

mod algorithm;
mod interfaces;

pub fn interior_point(
    objective_function: Vec<f64>,
    constraints: Constraints,
    initial_point: Vec<f64>,
    eps: usize,
    alpha: f64,
) -> Result<InteriorPoint, NotApplicableError> {
    let (n, m) = get_n_and_m(constraints).ok_or(NotApplicableError)?;

    if initial_point.len() != n + m
        || constraints
            .iter()
            .any(|row| row.0.len() != objective_function.len())
    {
        return Err(NotApplicableError);
    }

    let initial_point_is_feasible = constraints.iter().all(|(coefficients, sign, rhs)| {
        let constraint_sum: f64 = coefficients
            .iter()
            .zip(&initial_point)
            .map(|(coeff, x)| coeff * x)
            .sum();

        sign.compare(&constraint_sum, rhs)
    });

    if !initial_point_is_feasible {
        return Err(NotApplicableError);
    }

    Ok(InteriorPoint {
        done: false,
        x: DVector::from_vec(initial_point),
        big_a: build_big_a(constraints),
        c: DVector::from_vec(objective_function).resize_vertically(n + m, 0.),
        eps: up_to_n_dec_places(i32::try_from(eps).map_err(|_| NotApplicableError)?),
        alpha,
    })
}

fn get_n_and_m(constraints: Constraints) -> Option<(usize, usize)> {
    Some((constraints.len(), constraints.first()?.0.len()))
}

fn build_big_a(constraints: Constraints) -> DMatrix<f64> {
    let (n, m) = get_n_and_m(constraints).unwrap();

    let left_part_row_elements = constraints
        .iter()
        .flat_map(|(coefficients, _, _)| *coefficients)
        .copied();
    let right_part_diagonal_elements = &DVector::from_vec(
        constraints
            .iter()
            .map(|(_, sign, _)| match sign {
                Sign::Le => 1.,
                Sign::Eq => 0.,
                Sign::Ge => -1.,
            })
            .collect(),
    );

    let mut big_a =
        DMatrix::from_row_iterator(n, m, left_part_row_elements).resize_horizontally(m + n, 0.);

    big_a
        .view_mut((0, m), (n, m))
        .set_diagonal(right_part_diagonal_elements);

    let no_slack_rows = constraints
        .iter()
        .enumerate()
        .filter_map(|(i, (_, sign, _))| matches!(sign, Sign::Eq).then_some(i));

    let no_slack_columns = no_slack_rows.map(|j| m + j).collect::<Box<[_]>>();

    big_a.remove_columns_at(&no_slack_columns)
}

fn up_to_n_dec_places(n: i32) -> f64 {
    0.1_f64.powi(n) / 2.
}
```

crates/pt2-core/src/algorithm.rs

```rust
use na::{DMatrix, DVector};

use crate::interfaces::{Auxiliary, InteriorPoint, Iteration, NoSolutionError};

impl Iterator for InteriorPoint {
    type Item = Result<Iteration, NoSolutionError>;

    fn next(&mut self) -> Option<Self::Item> {
        if self.done {
            return None;
        }

        let size = self.x.len();

        let big_d = DMatrix::from_diagonal(&self.x);

        let big_a_tilde = &self.big_a * &big_d;
        let c_tilde = &big_d * &self.c;

        let big_p = {
            let big_i = DMatrix::identity(size, size);
            let big_a_tilde_tr = big_a_tilde.transpose();
            let Some(inverse) = (&big_a_tilde * &big_a_tilde_tr).try_inverse() else {
                self.done = true;
                return Some(Err(NoSolutionError));
            };
            big_i - big_a_tilde_tr * inverse * &big_a_tilde
        };
        let c_p = &big_p * &c_tilde;

        let Some(nu) = c_p
            .iter()
            .filter_map(|it| (it < &0.).then_some(it.abs()))
            .max_by(|a, b| a.partial_cmp(b).unwrap())
        else {
            self.done = true;
            return Some(Err(NoSolutionError));
        };
        let x_tilde = DVector::from_element(size, 1.) + (self.alpha / nu) * &c_p;

        let new_x = &big_d * &x_tilde;
        if (&new_x - &self.x).norm() < self.eps {
            self.done = true;
        }

        self.x = new_x;

        Some(Ok(Iteration {
            auxiliary: Auxiliary {
                big_d,
                big_a_tilde,
                c_tilde,
                big_p,
                c_p,
                nu,
                x_tilde,
            },
            decision_variables: self.x.clone_owned(),
            max: self.x.dot(&self.c),
        }))
        }
    }
```

crates/pt2-core/src/interfaces.rs

```
use na::{DMatrix, DVector};
use thiserror::Error;

#[derive(Error, Debug)]
#[error("method is not applicable")]
pub struct NotApplicableError;

#[derive(Error, Debug)]
#[error("problem has no solution")]
pub struct NoSolutionError;

pub struct Auxiliary {
    pub big_d: DMatrix<f64>,
    pub big_a_tilde: DMatrix<f64>,
    pub c_tilde: DVector<f64>,
    pub big_p: DMatrix<f64>,
    pub c_p: DVector<f64>,
    pub nu: f64,
    pub x_tilde: DVector<f64>,
}

pub struct Iteration {
    pub auxiliary: Auxiliary,
    pub decision_variables: DVector<f64>,
    pub max: f64,
}

pub struct InteriorPoint {
    pub(crate) done: bool,
    pub(crate) x: DVector<f64>,
    pub(crate) big_a: DMatrix<f64>,
    pub(crate) c: DVector<f64>,
    pub(crate) eps: f64,
    pub(crate) alpha: f64,
}

pub type Constraints<’a> = &’a [(&’a [f64], Sign, f64)];

pub enum Sign {
    Le,
    Eq,
    Ge,
}

impl Sign {
    pub fn compare<Lhs, Rhs>(&self, a: &Lhs, b: &Rhs) -> bool
    where
        Lhs: PartialOrd<Rhs>,
    {
        let cmp_function = match self {
            Self::Le => PartialOrd::le,
            Self::Eq => PartialEq::eq,
            Self::Ge => PartialOrd::ge,
        };

        cmp_function(a, b)
    }
}
```