Report 4

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: 4/5
- Team member 3: Asqar Arslanov (a.arslanov@innopolis.university), grade: 5/5
- Team member 4: Ilya-Linh Nguen (i.nguen@innopolis.university), grade: 4/5

Product Link

- https://github.com/quintet-sdr/optimization-pt4

Programming Language

- Rust (Cargo)
- To launch the code: `$ cargo run`

Tasks

Code:

- **src/main.rs**

```rust
use color_eyre::Result;

mod config;
mod tasks;

fn main() -> Result<()> {
    // Install panic hooks for pretty error messages.
    color_eyre::install()?;

    // Read the config file.
    let input = config::get()?;
    // Run the algorithms.
    tasks::solve(input);

    // Exit successfully.
    Ok(())
}
```

- **src/config.rs**

```rust
use std::fs;
use std::ops::Range;

use color_eyre::Result;
use serde::Deserialize;

pub fn get() -> Result<Config> {
    let raw = fs::read_to_string("input.toml")?;
    let parsed = toml::from_str(&raw)?;
    Ok(parsed)
}

#[derive(Deserialize)]
#[serde(rename_all = "kebab-case")]
#[allow(clippy::struct_field_names)]
pub struct Config {
    pub task_1: Task1,
    pub task_2: Task2,
    pub task_3: Task3,
}

#[derive(Deserialize)]
pub struct Task1 {
    pub interval: Range<f64>,
    pub tolerance: f64,
}

#[derive(Deserialize)]
pub struct Task2 {
    pub interval: Range<f64>,
    pub tolerance: f64,
}

#[derive(Deserialize)]
#[serde(rename_all = "kebab-case")]
pub struct Task3 {
    pub initial_guess: f64,
    pub learning_rate: f64,
    pub iterations: usize,
}
```

- **src/tasks.rs**

```rust
use colored::Colorize;

use crate::config::{Config, Task1, Task2, Task3};

mod bisection;
mod golden_section;
mod gradient_ascent;

pub fn solve(input: Config) {
    // Add empty lines between each task's output.
    task_1(&input.task_1);
    println!();
    task_2(input.task_2);
    println!();
    task_3(&input.task_3);
}

fn task_1(input: &Task1) {
    println!("Task 1");

    match bisection::solve_for(input.interval.clone(), input.tolerance) {
        Ok(root) => println!("root = {root}"),
        Err(root) => {
            let warning = format!(
                "Warning: f({}) = {} and f({}) = {} don't have opposite signs, so
the root should be invalid.",
                input.interval.start,
                bisection::f(input.interval.start),
                input.interval.end,
                bisection::f(input.interval.end),
            );
            println!("{}", warning.red());
            println!("root ?= {root}");
        }
    }
}

fn task_2(input: Task2) {
    println!("Task 2");

    let (x_min, f_of_x_min) = golden_section::solve_for(input.interval,
input.tolerance);

    println!("x_min = {x_min}, f(x_min) = {f_of_x_min}");
}

fn task_3(input: &Task3) {
    println!("Task 3");

    let (x_max, f_of_x_max) =
        gradient_ascent::solve_for(input.initial_guess, input.learning_rate,
input.iterations);

    println!("x_max = {x_max}, f(x_max) = {f_of_x_max}");
}
```

- **src/tasks/bisection.rs**

```rust
use std::ops::Range;

use tailcall::tailcall;

pub fn f(x: f64) -> f64 {
    (-6_f64).mul_add(x.powi(2), x.powi(3)) + 11_f64.mul_add(x, -6.)
}

pub fn solve_for(interval @ Range { start: a, end: b }: Range<f64>, eps: f64) ->
Result<f64, f64> {
    let root = actual_solve_for(interval, eps);

    if a <= b && f(a) * f(b) < 0. {
        // Signify that the root is valid.
        Ok(root)
    } else {
        // Signify that the root is probably invalid.
        Err(root)
    }
}

#[allow(unreachable_code)]
// Tail recursion optimization.
#[tailcall]
fn actual_solve_for(interval: Range<f64>, eps: f64) -> f64 {
    // Extract the ends of the interval to more convenient names.
    let Range { start: a, end: b } = interval;

    let c = (a + b) / 2.;

    if f(c).abs() < eps {
        return c;
    }

    let interval = if f(c).signum() == f(a).signum() {
        c..b
    } else {
        a..c
    };

    // Go to the next iteration.
    actual_solve_for(interval, eps)
}
```

- **src/tasks/golden_section.rs**

```rust
use std::cmp::Ordering;
use std::ops::Range;

use tailcall::tailcall;

fn f(x: f64) -> f64 {
    (x - 2.).mul_add(x - 2., 3.)
}

#[allow(unreachable_code)]
// Tail recursion optimization.
#[tailcall]
pub fn solve_for(interval: Range<f64>, eps: f64) -> (f64, f64) {
    /// $\frac{\sqrt{5} - 1}{2}$
    const FRAC_1_PHI: f64 = 0.618_033_988_749_894_8;

    // Extract the ends of the interval to more convenient names.
    let Range {
        start: x_l,
        end: x_r,
    } = interval;

    if x_r - x_l < eps {
        // Find the middle point between the interval ends.
        let middle = (x_l + x_r) / 2.;
        return (middle, f(middle));
    }

    let x_1 = FRAC_1_PHI.mul_add(x_l - x_r, x_r);
    let x_2 = FRAC_1_PHI.mul_add(x_r - x_l, x_l);

    let i = match f(x_1).total_cmp(&f(x_2)) {
        Ordering::Less => x_1..x_r,
        Ordering::Equal => x_1..x_2,
        Ordering::Greater => x_l..x_2,
    };

    // Jump to the next iteration.
    solve_for(i, eps)
}
```

- **src/tasks/gradient_ascent.rs**

```rust
use tailcall::tailcall;

fn f(x: f64) -> f64 {
    -x.powi(2) + 4_f64.mul_add(x, 1.)
}

fn f_prime(x: f64) -> f64 {
    (-2_f64).mul_add(x, 4.)
}

#[allow(unreachable_code)]
// Tail recursion optimization.
#[tailcall]
pub fn solve_for(x_0: f64, alpha: f64, n: usize) -> (f64, f64) {
    // When all iterations are complete.
    if n == 0 {
        return (x_0, f(x_0));
    }

    // Go to the next iteration.
    solve_for(alpha.mul_add(f_prime(x_0), x_0), alpha, n - 1)
}
```