We also used the R language for simple data manipulation and figures preparation – `rvest` [1] to download the information from Wikipedia, `ggmap` [2], `geosphere` [3] and `maps` [4] together with `ggplot` [5] and `ggrepel` [6] to work with maps and `dplyr` [7] and `stringr` [8] for easy data manipulations.

All tests are conducted on a Dell desktop with four $4.00GHz$ processors and $16Gb$ memory. The number of threads is set to be one. Using both formulation (three if we consider Dijkstra's algorithm) we obtain the following result:

- Travel distance from Clemson $\rightarrow$ Congaree = 132.96523411355977
- Travel distance from Congaree $\rightarrow$ Yellowstone = 1750.7684566034338
- Travel distance from Yellowstone $\rightarrow$ Yosemite = 661.182645802978

and the optimal route is as follows:
Clemson $\rightarrow$ Congaree $\rightarrow$ Gateway Arch $\rightarrow$ Wind Cave $\rightarrow$ Yellowstone $\rightarrow$ Yosemite.
The total travelling distance is 2544.9163365199715miles.

See the appendix for the code and results.

III. The Ellipsoid Method problem

As was mentioned in the project description, the ellipsoid method is one of the classical methods for solving convex optimization problems and, while it is not widely used in practice, it has significant theoretical value (in particular, it was used to prove that the linear programming is in class $\mathcal{P}$, i.e. polynomial-time solvable). The project topic was inspired by [9, §2.2][1] – however, due to high theoretical importance, there is more than extensive literature on the topic. As pointed out by Wolfe [10], first, an algorithm for convex nonlinear programming problems was proposed by the Soviet mathematicians Shor, Yudin and Nemirovsky (presented in [11]). It's modified version by Khachiyan (see, e.g. [12] for complete discussion on the topic.[2]. Also a review and an interpretation of Khachiyan's ideas were presented in [13]) was proved to be implementable in polynomial time, proving the polynomial-time solvability of linear programming. In addition to a numerous resources available online (including lecture notes by Dr. Stephen Boyd and many other excellent sources), a shorter treatment of the ellipsoid algorithm is provided in [9]; a more detailed overview of the algorithm, including some practical considerations (that appeared to be very useful for the implementation) as well as a detailed note on historical development of the method, can be found in [14].

In this project we implemented the Ellipsoid algorithm and tested in on several linear programming instances (including several specific cases). The rest of the section is organized as follows. We provide short notes on the problem formulation in the subsection III-A. Then we proceed to the algorithm high-level outline in the subsection III-B and consider several modification needed to account for "special" cases in the subsection III-C. The final algorithm outline (as it was implemented for this report) is given in the subsection III-D. Technical details of the implementation and software used is mentioned in the subsection III-E. Testing strategy is provided in the subsection III-F and is concluded by the results discussion in subsections III-G and III-H.

[1]we used the free version, available via `arxiv.org`

[2]The first note by Khachian appeared in the beginning of 1979

A. Problem formulation

In principle, the Ellipsoid algorithm (in particular, how it is presented in [9]) can be applied to any convex optimization problem. However, we designed version specifically for linear programming (LP) problems. Namely, we consider LP in the following form.

$$\begin{aligned} \min \quad & \vec{c}^T\vec{x} \\ S.t. \quad & A\vec{x} \le \vec{b} \end{aligned} \tag{P}$$

We deliberately do not impose any additional constraints to allow for the most general formulation. Notice, however, that popular formulations can represented in this form, viz.:

- **standard form** – constraints of the form $A\vec{x} = \vec{b}$ can be represented by the pair of $A\vec{x} \le \vec{b}$ and $-A\vec{x} \le -\vec{b}$, while nonnegativity constraints can be incorporated into the matrix A by adding $-\vec{x} \le \vec{o}$-type of constraints.
- **canonical form** (say, as put by [15]) – $A\vec{x} \ge \vec{b}$ can be transformed into $-A\vec{x} \le -\vec{b}$, nonnegativity constraints can be treated in the same way as above
- **maximization problems** – can be converted into minimization by the transformation $\max \vec{c}^T\vec{x} = -\min(-\vec{c})^T\vec{x}$,

and so on. Therefore, by adopting this formulation (P) we actually do not impose any additional constraints on the LP problem we consider. However, we do assume one constraint on the problem that is dictated by the logic of the algorithm – namely, we assume that:

1) there are at least one feasible point within a (reasonably *huge*) ball centered at the origin;
2) if there is an optimal point, it also lays somewhere within this ball mentioned above.

B. Key ideas

Following the logic of [9], let us start the description with the following Linear algebra introduction. Recall that an ellipsoid is a convex set, that can be expressed in the following form:

$$E = \{\vec{x} \in \mathbb{R}^n \mid (\vec{x} - \vec{o})^T H^{-1} (\vec{x} - \vec{o}) \le 1\}, \tag{1}$$

where, geometrically, $\vec{o} \in \mathbb{R}^n$ defines the center of the ellipsoid E and the $n \times n$ matrix H gives its shape – the semi-axes being given by the eigenvectors of H with the lengths given by the square root of the corresponding eigenvalues. The core (geometric) result that allows for the Ellipsoid method[3] can be formulated as follows:

Proposition 1: Let $E_0 = \{\vec{x} \in \mathbb{R}^n \mid (\vec{x} - \vec{o}_0)^T H^{-1} (\vec{x} - \vec{o}_0) \le 1\}$. Then: $\forall \vec{w} \in \mathbb{R}^n \setminus \vec{o}:\ \exists E_1$ – an ellipsoid, such that:

$$E_1 \supset \{\vec{x} \in E_0 \mid \vec{w}^T(\vec{x} - \vec{o}_0) \le 0\}, \mathrm{vol}(E_1) \le e^{-\frac{1}{2n}} \mathrm{vol}(E_0),$$

[3]that is presented in the Lemma 2.2 in [9]

that for $n \geq 2$ can be given by:

$$\begin{aligned} \vec{o}_1 &= \vec{o}_0 - \frac{1}{n+1}\frac{H_0\vec{w}}{\sqrt{\vec{w}^T H_0 \vec{w}}}, \\ H_1 &= \frac{n^2}{n^2-1}\big(H_0 - \frac{2}{n+1}\frac{H_0\vec{w}\vec{w}^T H_0}{\vec{w}^T H_0 \vec{w}}\big) \end{aligned} \quad \text{(E)}$$

Such ellipsoid given by (E) in fact represents the smallest ellipsoid that contains the whole half-ellipsoid from the proposition above. This fact was shown in [9, §2.2] by analyzing a simpler case of E_0 being a unit ball centered at the origin – and then noticing that any ellipsoid is just an affine transformation of a unit ball.

Now, the key idea of the algorithm, based on this geometric proposition 1 is essentially an extension of a bisection method and can be conceptually summarized as follows. Let us start with a (potentially, very large) ellipsoid that contains an optimal point of our optimization problem (P). If we could choose a "better" half of the ellipsoid (this being either containing a feasible set or containing a better feasible solution) – we would "slice" the ellipsoid in two parts, and chose the "better" one. An important fact here is that the volume of this (n-dimensional) ellipsoid would be decreasing exponentially if we continue in the same fashion. At some moment, we would reach the state when the "size" of the ellipsoid is comparable to the error margin (say, to the calculation precision limit given by the hardware constraints of a given computer device).

To illustrate the idea of the algorithm further, before delving into more technical (and rigorous) details, let us consider several typical steps of our algorithm. To be more specific, we will work with the following simplistic problem[4]:

$$\begin{aligned} \min \quad & -x_1 + 0.5x_2 & \\ S.t. \quad & -x_1 + x_2 & \leq 0 \\ & x_1 + x_2 & \leq 6 \\ & 1 \leq x_2 \leq 2 & \end{aligned}$$

The problem is two-dimensional, so we can easily analyze algorithm steps using graphs, such as those depicted in the figure 1 (x_1 is depicted along the horizontal axis, x_2 – along the vertical one). The feasible set is a trapezoid – shaded with blue, corners marked with large blue dots. The objective function is linear and the slope its isolines is 2 – isolines are depicted with dashed gray lines with objective increasing as we move towards right part of the figure. To illustrate the algorithm, let us omit the first 50 steps and assume for a moment that we start with the situation depicted in the Panel A. The logic of the ellipsoid method is as follows.

- at each step, our goal is essentially to find the next ellipsoid that will "better" than the previous one;
- note, that now the center of the ellipsoid (the red dot) is an infeasible point. So, let us separate the ellipsoid with a line[5] going through the center, that will be parallel to the constraint we have violated. In our case, this is the rightmost blue line of the trapezoid – so we make a cut as depicted in the Panel B with black solid line.
- using the formulae (E) we can switch to a new ellipsoid that will contain the part of the previous one that contains the feasible set *and* will be significantly smaller in volume than the previous one. The "old" ellipsoid is depicted in black dotted line (with the black dot as it's center) in the Panel B, and the "new" one – with red lines and red dot.
- The new ellipsoid center (Panel B, red dot) is a feasible point. Now, how do we choose which part of this ellipsoid we "like" more? We can just make a next cut correspond to an isoline of the objective function that goes through the ellipsoid center – and as long as the objective is linear, we will be sure that all the points in the right half of the ellipsoid will have a "better" (i.e. smaller) objective values. So, again, using the formulate (E) we will end up with a new ellipsoid depicted in the Panel C with red.
- interestingly, the central point in this algorithm version will not always be feasible. To see this, let us consider Panel C. The center is a feasible point, so we cut the ellipsoid with an isoline – and arrive at the Panel D, where the center of the new ellipsoid turned out to be outside the feasible region. This is not a huge problem, as we can just repeat the step that was described first above. But it implies that we cannot always rely on the last center point that we have seen – we need to keep track of feasible points and corresponding objective values.
- regardless of the fact that we sometimes get out of the feasible region, each time we decrease the volume of the ellipsoid and each time the ellipsoid contains an optimal point. So eventually we converge to this optimal point, as we can see in Panels E–H (notice that these are shown each five steps now).
- to complete the (conceptual) description, we need to mention the stopping criterion. We stop when the difference between the upper and lower bounds on the objective is small enough (which is a parameter of the algorithm – we denote it ϵ). The upper bound is given by the smalles value of the objective over all the feasible points that we have seen. To obtain the lower bound we follow what [14] called Sliding Objective Function Method. Modifying the formulae from this article for the minimization case, we take the lower bound to be:

$$\underline{\zeta_t} = \max\ \{\underline{\zeta_{t-1}}, \vec{c}^T\vec{x}_t - \sqrt{\vec{c}^T H_t \vec{c}}\}$$

C. A note on special cases and algorithm modifications

While the algorithm description so far seemed enough, its implementation and trying different test cases (more on that in the subsection III-F) implied the necessity to consider several specific cases.

1) Very large feasible region: Note that we need the very first ellipsoid to contain an optimal point. (Or, as we will see later, at least any feasible point). Somewhat following the spirit of [12], we are trying to ensure this by starting with reasonably large initial ball (say, with the radius of 1000 or 10000 with all the coefficients of the problem are bound by 10 or 100). While this, obviously, does not guarantee the presence of an optimal point (if it exists) in the initial ball –

[4]test case name `testcase-2d`, see the supplementary materials for details

[5]a hyperplane, in a general case

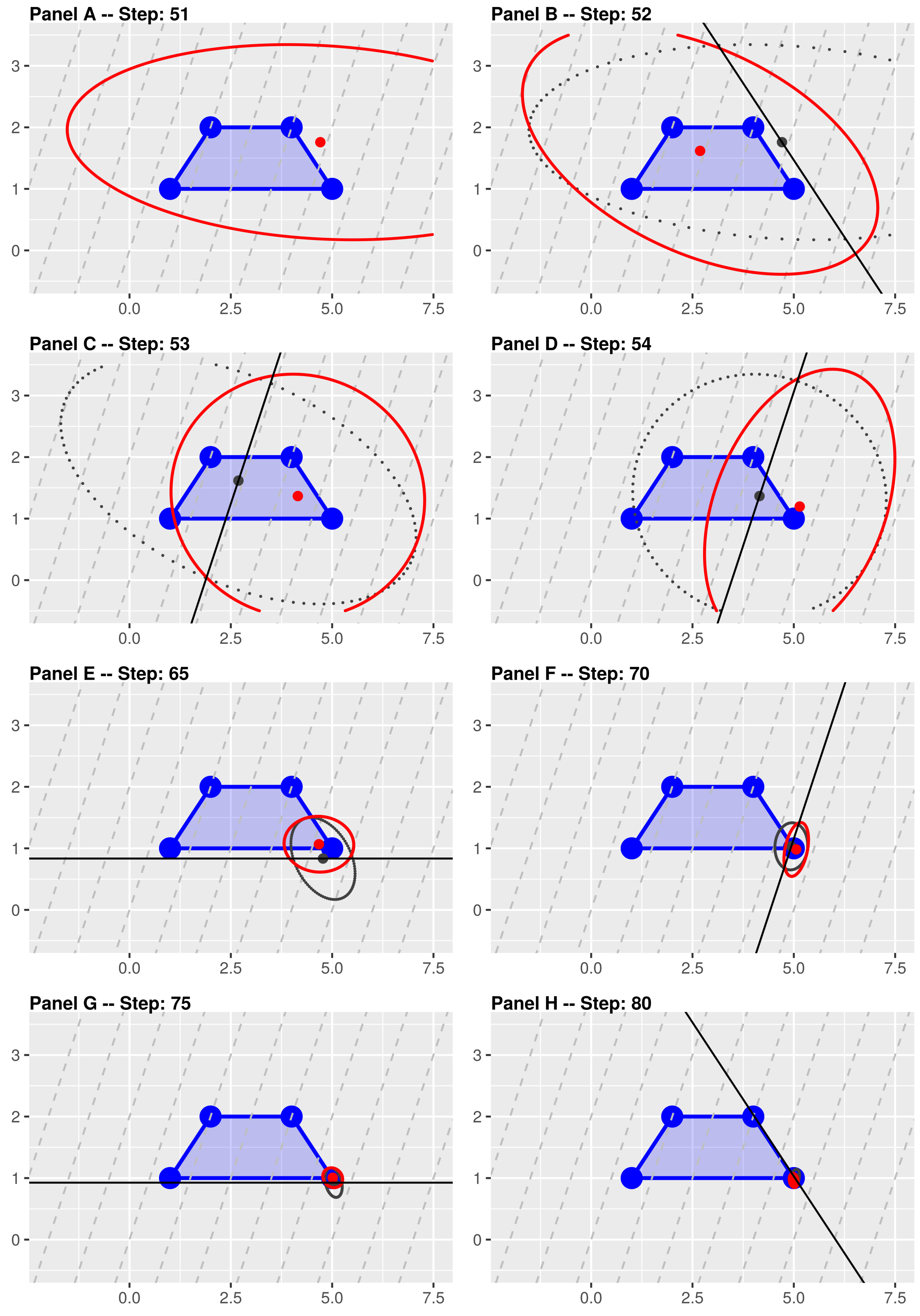

Fig. 1. Selected Ellipsoid algorithm steps

we consider it to be practically enough and never experienced any problems connected to the possibility of an optimal point being outside the initial ball. Theoretically, we could always choose the initial radius to be close to the largest number that can be represented by the given data type (say, `double` for `C++`, which appears to be `1.79769e+308` on our machine[6]). Although we have not implemented them, we would like to name possible alternative (or complementary) courses of action:

- re-scale the data for coefficients to be in a reasonable limits (as compared to the initial ball radius);
- check if the optimal point found is close to the given initial ball radius. If it is the case – we might want to start again with a larger radius (potentially, re-centering the ellipsoid);
- employ additional infeasibility check – say, solve dual problem in parallel (it obviously can be formulated algebraically from A, $\vec{b}$ and $\vec{c}$ – without the need to solve anything numerically). Note that if we find the dual to be unbounded – the primal will be known to be infeasible (and hence, no need in exploring larger initial ball radius). Solving dual and primal problems in parallel is discussed in more detail [14, §5].

2) Infeasible problem: An obvious special case is when the feasible set is empty. It is not clear immediately looking at the matrix A and the right-hand side $\vec{b}$. However, the ultimate purpose of the ellipsoid algorithm as it was proposed seemed to be exactly to find a feasible point. The logic behind that is we continue to "cross" ellipsoids with the hyperplanes corresponding to constraints, diminishing the volume. And if the volume drops below certain value[7] – we just disregard all the following iterations and conclude that the set is empty. Notice that, in principle, we can be wrong (say, if the feasible set is just one point) – and we will address some of such sub-cases below.

3) Non full-dimensional: A logical extension of the previous case is when the feasible set is not full-dimensional. For example, consider the following case:

$$\begin{aligned} \min \quad & x_1 + x_2 & \\ S.t. \quad & -x_1 + x_2 & \leq 0 \\ & x_1 - x_2 & \leq 0 \\ & x_1 \geq 1 & \end{aligned}$$

The feasible region is a ray, starting from the point $(1, 1)$, and the objective is set up in a way that $(1, 1)$ will be actually the optimal point (see the figure 2). But if we try to apply the algorithm implemented as described above[8] – we will end up in a strange behavior. Starting from a large enough initial ball centered at the origin, the algorithm will spend most of the time trying to find a feasible point ("jumping" around the line).

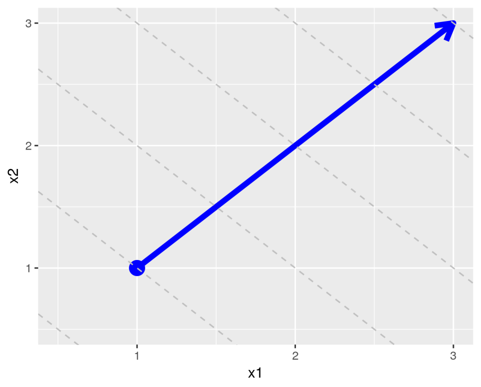

Fig. 2. A non-full dimensional problem example: the feasible set is a ray, emanating from the point $(1, 1)$

And by the time it will find one, the volume of the ellipsoid will be already small enough to stop (recall that it decreases exponentially). So we will end up with a strange feasible point somewhere far away along the ray – namely, $x_1 = x_2 =$ `2.15136321645451e+06` for our test case. An illustration of the fact that the "straightforward" implementation of the Ellipsoid method might not converge to the feasible set if it has zero volume also provided in [14, Appendix C].

To alleviate this effect we implemented the following logic. Let us consider certain small[9] violation of constraints as acceptable. Therefore we virtually create "thickness" (i.e. volume) of our feasible set – even if it does not possess any, strictly speaking. This approach essentially by introduced by Khachiyan and described, say, in [12], [13] – we mostly considered the description from [14, §2].

If we modify the algorithm in this simple way[10] the algorithm correctly finds the optimal point $(1, 1)$ for the example above.

4) Unbounded problem: Another interesting special case is when the problem is unbounded. Following [14, §2], we implemented the following approach. We note that an LP problem is unbounded if and only if there is a direction (i.e. ray) such that all the points of this ray are feasible and the objective value along this ray is unbounded. That is, there exists some vector $\vec{d}$ such that:

$$\begin{aligned} A\vec{d} &\leq \vec{o} \\ \vec{c}^T\vec{d} &< 0 \end{aligned} \tag{U}$$

That is, we can increase the "length" of $\vec{d}$ indefinitely – it will not affect feasibility (if we start from some feasible point – i.e. consider some point $\vec{x} = \vec{x}_0 + \vec{d}$, where $\vec{x}_0$ is feasible), but will allow for arbitrary small objective function. We also notice that we can safely rewrite the last constraint as $\vec{d} \leq -1$ (as $\vec{d}$ satisfying (U) exists if and only if there exists another $\vec{d\prime}$ satisfying this "simplified" system). So, checking for unboundedness can be reduced to testing if the set given by (U) is empty or not – that can be done by

[6]the result was obtained using a simple code that was available online as a supplementary material for [16]

[7]that depends on the representation of numbers in the hardware – but can in principle be just approximated, empirically

[8]note that to test this you will need to compile the solver with `-DSET_THICKNESS=0` flag – or, using our makefile, do `make compile-thin` and then run the test case as `./testell data/testcase-NFD.data -verbose > nonfull.log`

[9]given by `SET_THICKNESS` parameter in our source code

[10]for the parameter value `SET_THICKNESS=1e-8`

the ellipsoid algorithm. That is, we just add a procedure of "unboundedness-check" before the main solution procedure.

5) Other modifications: As pointed out in [14, §6], due to numerical issues errors can accumulate during calculation of ellipsoids resulting in poor performance (that is, earlier stopping and/or "convergence" to "wrong" points). To alleviate these effects, instead of using the simple ellipsoid update rule (E) we used[11] one provided in [14] (actually, in the form provided in the original paper [12] (formulae (3.7)):

$$\vec{x}_{k+1} \approx \vec{x}_k - \frac{B_k \eta_k}{(n+1)\|\eta_k\|}$$

$$B_{k+1} \approx \left(1 + \frac{1}{16n^2}\right)\frac{n}{\sqrt{n^2-1}}\left\{B_k + \left[\sqrt{\frac{n-1}{n+1}} - 1\right]\frac{B_k \hat{\eta}_k}{\|\eta_k\|^2}\right\},$$

where the vector $\eta_k \leftrightharpoons B_k' \vec{w}_k$ (recall that $\vec{w}_k$ is a hyperplane from the proposition 1) and the matrix $\hat{\eta}$ is constructed from this vector as follows: $\hat{\eta}_{\alpha\beta} \leftrightharpoons (\eta_k)_\alpha \cdot (\eta_k)_\beta$.

D. The algorithm outline

Having discussed the ideas above, we are ready to outline the algorithm more formally now:

E. Implementation details

1) Platform and tools choices: We implementd the whole project in the Linux environment, using a laptop with Ubuntu 18.04.1 LTS system. This allowed for certain flexibility in working with log files and tools choice.

The algorithm was implemented in C++ and compiled with the standard `g++` compiler. We used the following libraries:

- `Armadillo` to allow for easy manipulation with vectors and matrices [17] (as well as `BLAS`);
- `Gurobi` library and solver for test cases generation (to find 100%-correct answers), [18];
- `Boost` C++ library (say, [19]) – mostly to parse command-line options efficiently for test cases generator;
- standard `libm` math library for C++;

Build recipes (i.e. libraries to compile with, etc.) were written for the standard GNU `make` tool. Some "glue" for automating testing procedures and parsing log files was added with standard `bash` shell-scripts.

Detailed log-files analysis and figures preparation was done in R [20], using the following packages:

- `readr` [21] and `stringr` [8] for data parsing (including regular expressions);
- `ggplot2` [5], `ggrepel` [6], `gridExtra` [22] and `car` [23] for (an excellent!) figures preparation routines;
- `dplyr` [7] for convenient data-manipulations;

Some early exploratory work and simple numerical experiments was done with `Matlab`.

[11]we actually implemented both variants, and the "plain" ellipsoid update rule can be turned on if the client is run with `-verbose -plain` flags

Algorithm 1 Finds an optimal point for the problem (P), if it belongs to the ball of radius R_M – or classifies the problem as infeasible or unbounded. Interrupts after T_{max} steps

```
procedure SOLVE($A$,$\vec{b}$,$\vec{c}$, $T_max$)
    Check if the problem is unbounded (sec. III-C4)
    if isUnbounded($A, \vec{b}, \vec{c}$) then
        return "UNBOUNDED";
    end if                                          ▷ initialization
    status ← IS_INFEASIBLE;
    bestObjective ← $+\infty$;                      ▷ keeping track...
    bestPoint ← $\varnothing$;                      ▷ ...of the best point
    $\vec{w} \leftarrow \varnothing$;               ▷ cutting plane
    step ← 0;
    LB ← $+\infty$;                                 ▷ lower bound
    $E \leftarrow \{x \ : \ \|x\| \le R_M\}$;[12]
    repeat
        if centerIsFeasible($E$,$A$,$\vec{b}$) then
            if $\vec{c}^T o_E <$ bestObjective then
                bestObjective ← $\vec{c}^T o_E$;
                bestPoint ← $o_E$;
                status ← IS_FEASIBLE;
            end if
            $\vec{w} \leftarrow \vec{c}$;                ▷ correspond to costs
        else                                        ▷ infeasible center
            $i$ ← findViolatedRow($A, \vec{b}, o_E$);
            $\vec{w}^T \leftarrow \vec{a}^{T_i}$;
        end if
        $E$ ← updateEllipse($\vec{w}, E$);
    until vol($E < \epsilon$ or stopCriterion or step $\ge T_{max}$
    if status==IS_FEASIBLE then
        return IS_OPTIMAL, $o_E$
    end if
end procedure
```

```
procedure ISUNBOUNDED($A, \vec{b}, \vec{c}$)        ▷ Apply
the core ellipsoid algorithm above to find "unboundedness
direction" – see the sec. III-C4
end procedure
```

```
procedure CENTERISFEASIBLE(Ellipsoid $E$, $A$, $\vec{b}$)
    Calculate the differences vector $\vec{r} \leftrightharpoons A\vec{x} - \vec{b}$;
    if $\max(r_i)$ < SET_THICKNESS then             ▷ see III-C3
        return TRUE;
    else
        return FALSE;
    end if
end procedure
```

```
procedure FINDVIOLATEDROW($A, \vec{b}, \vec{x}$)    ▷ finds the first
row that violates $\vec{a}_i^T \vec{x} \le b_i$
end procedure
```

```
procedure STOPCRITERION                             ▷ see sec. III-B
    LB ← $\max\{LB, \vec{c}^T o_E - \sqrt{\vec{c}^T H \vec{c}}\}$;
    UB ← bestObjective;
    return (UB - LB) ≤ $\epsilon$
end procedure
```

2) A note on the source code structure: The code is logically split into several parts – let us briefly sketch the structure in this section.

- **Algorithm code** – the main algorithm implementation is placed into `ellipsoid.h` source file and organized into two key classes: a base `class LPSolver`, that describes key parameters of the solver (such as data structure needed for the model, etc.) and a derived `class EllipsoidSolver`, in which the key algorithm logic is implemented in the following functions with (hopefully) self-explanatory names: `solve()`, `isUnbounded()`, `checkFeasibility()`, `updateEllipseKhachiyan()`;
- **Client code** – needed to run the algorithm on the test cases (read the data, pass it to our solver, etc.) – implemented in `ell-test.cpp` file.
- **Auxiliary code** – needed mostly for the test cases generation. Implemented in the `generate_tests.cpp`. Also there is a small program to show `double` maximum value called `lims.cpp` and the R source code needed to produce figures in the `vlogParser.R`.
- **Bash shell-scripts** `run_tests.sh` and `run_kmc_tests.sh` are needed to run all the prepared test-cases with the compiled solver, and then summarize the results into a tables from individual log-files.

F. Test cases and testing strategy

We implemented algorithm testing in three tiers as follows.

In Tier-1 tests (that we called "Pre-defined test cases" we tried to construct several typical situations described in the section III-C. We also tried a "special" cases when there is an optimum *line segment* and unbounded feasible sets (although, finite optimal solutions). We created these "basic" test cases in two dimensions, for simplicity – however, added two more tests to try the algorithm for higher dimensions. Test case information is summarized in the table II.

Then, to test the robustness of the algorithm and ability to handle various feasible/unbounded situations, we generated 300 more "Tier-2" test cases as follows:

- 50 random 5x10[13] test cases with "free variables";
- 50 random 5x10 test cases with non-negative variables[14]
- 50 random 10x10 "free-variable" test cases;
- 50 random 10x10 "non-negative" test cases;
- 50 random 15x10 "free-variable" test cases;
- 50 random 15x10 "non-negative" test cases;

We tried to imply as few restrictions on test instances as possible[15] – however, we did enforced certain scale limits

[13]meaning $m \times n$

[14]in this list, number of non-negativity constraints was **not** included into number of constraints – i.e., for this particular case in fact these were 15x10 instances

[15]so if there are any problems with the implementation – they show up. And indeed, this allowed for efficient debugging

TABLE I
COEFFICIENTS GENERATION LIMITS

Coefficients	Limits
A_{ij}	$-10, \dots, 10$
c_i	$-10, \dots, 10$
b_i	$-100, \dots, 100$

TABLE II
PRE-DEFINED TEST CASES SUMMARY

Case name	n	m[a]	Optimum	Comment
`testcase-2d`	2	4	finite	A simple "trapezoid"
`testcase-2d-UBR`	2	3	finite	Unbounded feasible set
`testcase-2d-UBLine`	2	3	finite	Unbounded, opt. line segment
`testcase-IF`	2	2	infeasible	Simple case
`testcase-NFD`	2	3	finite	Actually, 1-D ray[b]
`testcase-UB`	2	4	∞	Unbounded problem
`testcase-5d`	5	$4+5$	finite	Higher-dimensional[c]
`testcase-20d`	20	$20+20$	finite	Even higher dimensions

[a] two numbers with $+$ between them denote number of constraints of the form $A\vec{x} \leq \vec{b}$ and number of non-negativity constraints, respectively
[b] NFD stands for Non-Full Dimensional set [c] represents a (little scaled) problem no. 4 from the home assignment no. 8

to alleviate potential numerical issues[16]. Namely, coefficients were generated from uniform random distribution, within the limits that are specified in the table I.

Each test case instance contained all the data necessary to run the algorithm (viz., A matrix, vectors $\vec{b}$ and $\vec{c}$) as well as the "correct answer" (problem status – whether it has a finite optimum, is unbounded or infeasible along with an optimal point, if it exists).

Finally, to test the algorithm performance against a special case (that is known to take exponential time for simplex algorithm) we engineered 10 more "Tier-3" specialized test cases that are described in details below, in the section III-G1

G. Testing result and performance

Tests results are provided in the supplementary materials (see the section VI-B1). The Ellipsoid algorithm was able to find a solution close enough to optimum for every test case, but it seems like it did not always finish in reasonable time within the "correct" point (we classified tests where the found value was more than 0.01 away from the real optimal objective as `FAILED`). We summarize tests results in the table III. We have not (directly) benchmarked the Ellipsoid algorithm in terms of performance against, say, simplex method for the sake of keeping the report (and the project) reasonably short, but we think that the robustness of the method seems to be acceptable (although, it might still require some more careful treatment for selected cases – especially non fully-dimensional sets).

1) Performance on Klee-Minty cubes: It might be of special interest to consider the performance of the algorithm on a specially structured problems – namely, so called Klee-Minty cubes [25]. As we have seen in the class, the simplex algorithm with a simple pivoting rule (called "Dantzig rule", choosing

[16]Though, it might be an interesting idea for a separate, say, next-year project. To create a robust LP instance generator that will allow to generate feasible, infeasible, optimal (+ option of totally unimodular matrices?) problems with given parameters. It might be implemented starting from a simplex tableau, or with some geometric considerations – e.g., see [24]. Unbounded problems can be generated from infeasible duals...

TABLE III
TEST RUNS SUMMARY

Test result	No. of tests
Completely correct	304
Wrong status[a]	0
Objective deviation more than 0.01	4
TOTAL:	**308**
Including:	
Finite-optimum cases:	17
Infeasible cases:	70
Unbounded cases:	221

[a] i.e. making a mistake classifying the problem as having a finite optimum, unbounded or infeasible

the "most negative" reduced cost) when tested against these special problems tends to visit each vertex, that is, take exponential number of steps in the problem size. We will stick with the problem formulation provided in the Wikipedia article[17]:

$$\begin{array}{lllllll} \max & 2^{D-1}x_1 & +2^{D-2}x_2 & +\dots & +2x_{D-1} & +x_D & \quad (2)\\ S.t. & x_1 & & & & & \le 5\\ & 4x_1 & +x_2 & & & & \le 25\\ & 8x_1 & +4x_2 & +x_3 & & & \le 125\\ & & & \vdots & & & \\ & 2^D x_1 & +2^{D-1} & +\dots & +4x_{D-1} & +x_D & \le 5^D \end{array}$$

That is, assuming that we consider a D-dimensional Klee-Minty cube, the costs vector is two to the powers from zero to $(D-1)$, and coefficients matrix is lower triangular, with nonzero coefficients being 2 to certain power, starting from 2 immediately below the main diagonal and increasing to the left. The main diagonal itself is just a D-unit vector.

It seems like one of the key messages and key results behind the story by Khachiyan was the proof that the linear programming is actually in the $\mathcal{P}$ class (to the point of having a titles like "A Soviet Discovery Rocks the World of Mathematics" on the front pages of magazines[18]). So we wanted to illustrate this fact somehow.

To do this, we generated ten Klee-Minty cube test cases, from the degree[19] $D = 3$ to 13 – they are summarized in supplements (see section VI-B2; as we can see, the algorithm successfully finds the correct optimal vertex). Now, how we can see that an algorithm runs in polynomial time? Assume it is $O(D^k)$. But then, theoretically, if we plot $log(D)$ along one axis and $log(T)$ (where T is the number of steps that took the algorithm to solve the problem) along the other one – we should see a straight line. The result from our test cases is presented in the figure 3. It looks exactly as a straight line (if the number of steps was exponential – say, 2^D, as we observed in one of our assignments – we should have seen some nonlinear behavior, like $\log(2^D)$). While the result, obviously, cannot be treated as a proof of algorithm polynomial complexity (that was provided in several original papers we mentioned above), we hope that it provides a nice illustration of this known fact.

[17]we were not able to access the original Feb 1970 paper
[18]no jokes, [26]
[19]we thought that generating 1D and 2D cases will not be very interesting

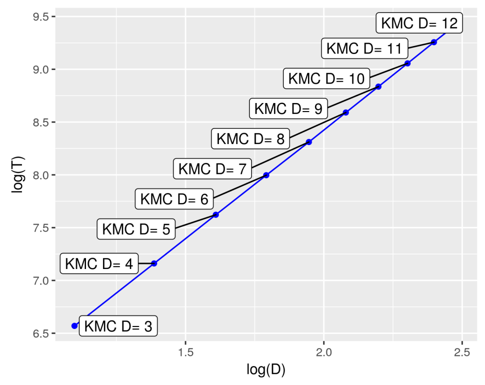

Fig. 3. Algorithm performance on Klee-Minty cubes – log(number of steps) vs log(problem dimensionality)

H. Conclusion

This part of the project was focused on one specific variant of the ellispoid algorithm. We implemented the algorithm in C++ along with some bare minimum testing infrastructure (command-line "client" application for testing, various test cases generator and some auxiliary scripts with `bash` and R-visualization). We briefly checked the robustness of the algorithm with 300 random test cases and specifically looked at the performance of the algorithm on Klee-Minty cubes. There are several implementations of the Ellipsoid algorithm (that are also discussed, e.g. in [14] and other sources). There are, no doubt, several ways in which this algorithm can be improved[20]. There are even not complicated modifications that are known to perform better (say, deep cuts – we can gain some efficiency if we relax the necessity to make a cut via the ellipsoid center). It might be interesting also to implement a simplex algorithm to compare them explicitly on Klee-Minty cubes. Although what we have *not* implemented concerning the ellipsoid method might constitute another project (if not course), we humbly hope that it was a suitable illustration of the method and its key performance characteristics.

IV. SVM Model for Iris Data and Handwriting Digit

A. Support Vector Machines Background

Support vector machines (SVM) are a class of very powerful, and highly flexible modeling techniques. Originally developed for classification; then extended to predictive regression. SVM is not limited to only linear models.

B. From Linear SVM to Non-linear SVM

In class, we discussed the simple linear SVM (Hard-margin and Soft-margin). Although, for the digit recognizing

[20]for example, a very logical step would be to run the same program with dual problem in parallel – and as long as the code takes only one thread and nearly all contemporary machines have several cores, we could obtain a more reliable stopping criterion (due to strong duality of LP) at no wall-clock time cost

References

[1] H. Wickham, *rvest: Easily Harvest (Scrape) Web Pages*, 2016, r package version 0.3.2. [Online]. Available: https://CRAN.R-project.org/package=rvest

[2] D. Kahle and H. Wickham, "ggmap: Spatial visualization with ggplot2," *The R Journal*, vol. 5, no. 1, pp. 144–161, 2013. [Online]. Available: http://journal.r-project.org/archive/2013-1/kahle-wickham.pdf

[3] R. J. Hijmans, *geosphere: Spherical Trigonometry*, 2017, r package version 1.5-7. [Online]. Available: https://CRAN.R-project.org/package=geosphere

[4] O. S. code by Richard A. Becker, A. R. W. R. version by Ray Brownrigg. Enhancements by Thomas P Minka, and A. Deckmyn., *maps: Draw Geographical Maps*, 2018, r package version 3.3.0. [Online]. Available: https://CRAN.R-project.org/package=maps

[5] H. Wickham, *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2009. [Online]. Available: http://ggplot2.org

[6] K. Slowikowski, *ggrepel: Automatically Position Non-Overlapping Text Labels with 'ggplot2'*, 2018, r package version 0.8.0. [Online]. Available: https://CRAN.R-project.org/package=ggrepel

[7] H. Wickham, R. Francois, L. Henry, and K. MÃijller, *dplyr: A Grammar of Data Manipulation*, 2017, r package version 0.7.4. [Online]. Available: https://CRAN.R-project.org/package=dplyr

[8] H. Wickham, *stringr: Simple, Consistent Wrappers for Common String Operations*, 2018, r package version 1.3.0. [Online]. Available: https://CRAN.R-project.org/package=stringr

[9] S. Bubeck, *Convex Optimization: Algorithms and Complexity*. now, 2015. [Online]. Available: https://ieeexplore.ieee.org/document/8187196

[10] P. Wolfe, "Some references for the ellipsoid algorithm," *Management Science*, vol. 26, no. 8, p. 747âĂŞ749, Aug 1980.

[11] N. Z. Shor, "Cut-off method with space extension in convex programming problems," *Cybernetics*, vol. 13, no. 1, p. 94âĂŞ96, Jan 1977.

[12] L. Khachiyan, "Polynomial algorithms in linear programming (in russian)," *Journal of Computational Mathematics and Mathematical Physics (in Russian)*, vol. 20, no. 1, p. 51âĂŞ68, 1980.

[13] P. GÃącs and L. LovÃąsz, *KhachiyanâĂŹs algorithm for linear programming*, ser. Mathematical Programming Studies. Springer Berlin Heidelberg, 1981, p. 61âĂŞ68. [Online]. Available: https://doi.org/10.1007/BFb0120921

[14] R. G. Bland, D. Goldfarb, and M. J. Todd, "Feature articleâĂŤthe ellipsoid method: A survey," *Operations Research*, vol. 29, no. 6, p. 1039âĂŞ1091, Dec 1981.

[15] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali, *Linear Programming and Network Flows*, 4th ed. Wiley, Dec 2009.

[16] D. R. Stephens, C. Diggins, J. Turkanis, and J. Cogswell, *C++ Cookbook: Solutions and Examples for C++ Programmers*, 1st ed. OâĂŹReilly Media, Nov 2005.

[17] C. Sanderson and R. Curtin, "Armadillo: a template-based c++ library for linear algebra," *The Journal of Open Source Software*, vol. 1, no. 2, p. 26, Jun 2016.

[18] L. Gurobi Optimization, "Gurobi optimizer reference manual," 2018. [Online]. Available: http://www.gurobi.com

[19] B. Karlsson, *Beyond the C++ Standard Library: An Introduction to Boost*, 1st ed. Addison-Wesley Professional., Aug 2005. [Online]. Available: http://www.informit.com/store/beyond-the-c-plus-plus-standard-library-an-introduction-9780321133540

[20] R Core Team, *R: A Language and Environment for Statistical Computing*, 2018, r Foundation for Statistical Computing. [Online]. Available: https://www.R-project.org

[21] H. Wickham, J. Hester, and R. Francois, *readr: Read Rectangular Text Data*, 2018, r package version 1.2.1. [Online]. Available: https://CRAN.R-project.org/package=readr

[22] B. Auguie, *gridExtra: Miscellaneous Functions for "Grid" Graphics*, 2017, r package version 2.3. [Online]. Available: https://CRAN.R-project.org/package=gridExtra

[23] J. Fox and S. Weisberg, *An R Companion to Applied Regression*, 2nd ed. Thousand Oaks CA: Sage, 2011. [Online]. Available: http://socserv.socsci.mcmaster.ca/jfox/Books/Companion

[24] E. Castillo, R. E. Pruneda, and M. Esquivel, "Automatic generation of linear programming problems for computer aided instruction," *International Journal of Mathematical Education in Science and Technology*, vol. 32, no. 2, p. 209âĂŞ232, Mar 2001.

[25] V. Klee and G. J. Minty, *HOW GOOD IS THE SIMPLEX ALGORITHM,*, Feb 1970, no. TR-22. [Online]. Available: https://apps.dtic.mil/docs/citations/AD0706119

[26] M. W. Browne, "A soviet discovery rocks world of mathematics," *The New York Times*, Nov 1979. [Online]. Available: https://www.nytimes.com/1979/11/07/archives/a-soviet-discovery-rocks-world-of-mathematics-russians-surprise.html

[27] V. S. Cherkassky and F. Mulier, *Learning from data: Concepts, theory, and methods*. John Wiley and Sons, 2006.

[28] Y. B. Dibike, S. Velickov, and D. Solomatine, "Support vector machines: Review and applications in civil engineering," in *Proc. of the joint workshop on Applications of AI in Civil Engineering, Cottbus-2000, Germany*. Citeseer, 2000.

[29] V. Cherkassky and Y. Ma, "Selecting of the loss function for robust linear regression," *Neural computation*, 2002.

[30] Y. B. Dibike, S. Velickov, D. Solomatine, and M. B. Abbott, "Model induction with support vector machines: introduction and applications," *Journal of Computing in Civil Engineering*, vol. 15, no. 3, pp. 208–216, 2001.

[31] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statistics and computing*, vol. 14, no. 3, pp. 199–222, 2004.

[32] T. Yoshioka and S. Ishii, "Fast gaussian process regression using representative data," in *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, vol. 1. IEEE, 2001, pp. 132–137.

[33] Y. MATSUMURA, "Mnist to csv convertion tool." [Online]. Available: https://github.com/ymattu/fashion-mnist-csv/blob/master/make_data.R

[34] S. Simha, "Processing svm in r1." [Online]. Available: https://github.com/Srungeer-Simha/MNIST-digit-recognition-using-SVM/blob/master/MNIST\%20digit\%20recognition\%20-\%20SVM.R

[35] A. Marjani, "Processing svm in r2." [Online]. Available: https://api.rpubs.com/dardodel/digit_recognition_KNN_NN_SVM

[36] Unkown, "Processing svm in r3." [Online]. Available: https://http://euler.stat.yale.edu/~tba3/stat665/lectures/lec11/script11.html