programming_exercise_2

November 23, 2022

1 Programmierübung 2 zu *Grundlagen der Optimierung* (WS2021)

1.1 Einführung

1.1.1 Verantwortlich

- Prof. Dr. Roland Herzog
- Dr. Georg Müller
- M.Sc. Masoumeh Hashemi

Die Webseite dieser Veranstaltung finden Sie unter https://tinyurl.com/scoop-gdo.

1.1.2 Zielsetzung

Das Ziel dieses *Jupyter Notebooks* ist es, Ihnen das Verhalten des Simplex-Algorithmus aus Kapitel 2 des Skripts zur Lösung linearer Optimierungsprobleme nahezubringen. Wir werden 1. Das Simplex Verfahren mit zugehöriger Phase I und verschiedener Index-Auswahlverfahren implementieren 1. Die Implementierung testen 1. Den Einfluss zweier verschiedener Indexauswahlregeln auf den Simplex-Algorithmus untersuchen 1. Den Einfluss von der Wahl der Toleranzen im Simplex-Algorithmus untersuchen 1. Die Performance des Simplex-Algorithmus untersuchen 1. Tschebyschow-Zentren berechnen und visualisieren

1.1.3 Zur Bedienung des Notebooks

Bitte schauen Sie hier unter dem Punkt "Bedienung" für mehr Informationen zur Nutzung und Bedienung des Notebooks.

1.2 Implementierung des Simplex-Verfahrens für lineare Programme

In diesem Abschnitt implementieren wir das Simplex-Verfahren- den Kern dieser Programmierübung. Bevor wir mit dem Simplex selbst beginnen können, benötigen wir jedoch Auswahlverfahren für die Indizes, die anhand der reduzierten Kosten in die Basis aufgenommen werden bzw. anhand des Quotiententests aus der Basis entfernt werden. Konkret wollen wir die folgenden Regeln implementieren: 1. Die Auswahlregel nach Bland für den in die Basis aufzunehmenden Index 1. Die Auswahlregel nach Bland für den aus der Basis zu entfernenden Index 1. Die Steepest-Descent-Auswahlregel, für den in die Basis aufzunehmenden Index, welche den Index mit dem kleinsten Eintrag im reduzierten Kostenvektor zurückgibt 1. Die Auswahlregel nach Bland für den in die Basis aufzunehmenden Index

Aufgabe: Vervollständigen Sie den Code und führen Sie die Zelle aus.

[ ]:
```python
# This module implements index selection rules for the primal simplex method

import numpy as np

def blands_entering_rule(reduced_cost, tolerance):
        """
        Accepts:
                        reduced_cost: the reduced cost vector of full length
                           tolerance: a tolerance on when to consider a reduced␣
 ↪cost "actually" negative
        Returns:
                        The lowest index with corresponding negative reduced␣
 ↪cost entry
        """
        ### TODO BEGIN ###
        # return index according to bland
        ### TODO END ###

def blands_leaving_rule(restrictions):
        """
        Accepts:
                        restrictions: the indices corresponding to the␣
 ↪halfspaces restricting further movement along the update direction. These␣
 ↪components will be zero in the iterate after the update was performed.
        Returns:
                        The lowest index that restricts further movement
        """
        ### TODO BEGIN ###
        # return index according to bland
        ### TODO END ###

def steepest_descent_entering_rule(reduced_cost, tolerance):
        """
        Accepts:
                        reduced_cost: the reduced cost vector of full length
                           tolerance: unused
        Returns:
                        The index with lowest corresponding negative reduced␣
 ↪cost entry
        """
        ### TODO BEGIN ###
        # return index according to steepest descent
        ### TODO END ###
```

Jetzt, wo wir die Auswahlregeln zur Hand haben, folgt die Implementierung des Simplex-Algorithmus.

Aufgabe: Vervollständigen Sie den Code und führen Sie die Zelle aus.

[ ]:
```
# This module implements the primal simplex method for solving linear programs
# It consists of two functions: The "actual" simplex method in␣
 ↪run_primal_simplex_method,
# which expects a basis for a initial feasible basic vector as input, and the␣
 ↪wrapper
# primal_simplex_method, which will do a phase I if no initial basis was␣
 ↪provided for
# the original problem.

import numpy as np

def primal_simplex_method(A, b, c, B = None, entering_rule =␣
 ↪blands_entering_rule, leaving_rule = blands_leaving_rule, parameters={}):
        """
        Solve a linear optimization problem in standard form, i.e., a problem of
        the form
                min c*x
                s.t. Ax = b, x >= 0,
        using the primal simplex method with Bland's rule.

        This function wraps the actual simplex in order to include a␣
 ↪phase-I-search, if
        no initial feasible basic point is provided.

        Accepts:
                                c: the linear cost's vector
                                A: the constraint's matrix
                                b: the constraint's right-hand-side vector
                                B: basis indices for initial feasible basic vector
                                   defaults to None, in which case a basis will be␣
 ↪determined from a Phase I
                        entering_rule: the rule that selects the index that enters the␣
 ↪basis from the reduced cost vector
                         leaving_rule: the rule that selects the index that leaves the␣
 ↪basis from the quotient vector
                           parameters: optional parameters (dictionary);
                                       the following key/value pairs are evaluated:
                                       ["max_iterations"]: maximum number of iterations
                                                  ["tol"]: tolerance for optimality␣
 ↪check on reduced cost vector entries and feasibility in phase I
                                             ["verbosity"]: "verbose" or "quiet"
                                        ["keep_history"]: whether or not to store the␣
 ↪iteration history (0, 1 or 2)

        Returns:
                result: a dictionary containing the key/value pairs
```

```
                        solution: final iterate
                        function: the final iterate's objective value
                    basis_indices: the final iterate's basis indices
                       iterations: number of iterations performed
                        exitflag: flag encoding why the algorithm terminated
                                  0: optimal solution found
                                  1: problem is unbounded
                                  2: maximum number of iterations reached
                                  3: problem is infeasible (In which case␣
↪all other fields of result will be None)
        """
        # Check some parameter requirements and set to default if not provided
        verbosity = parameters.get("verbosity", "quiet")
        tol = parameters.get("tol", 0)

        # If initial basis for feasible basic vector was not provided, find one␣
↪in phase I
        if B is None:
                # Dump some output
                if verbosity == 'verbose':
                        print("No initial basis provided. Starting phase I.")

                # Get problem dimensions
                m, n = A.shape


                ### TODO BEGIN ###
                # Generate input data for the phase-I-problem
                A_I = ...
                b_I = ...
                c_I = ...
                B_I = ...
                ### TODO END ###

                # Solve phase-I-problem
                phase_I_result = run_primal_simplex_method(A_I, b_I, c_I, B_I,␣
↪entering_rule, leaving_rule, parameters)

                # Check if phase I terminated correctly
                if phase_I_result["exitflag"] == 2:
                        raise BaseException("Phase I terminated because the␣
↪maximum number of iterations were performed.")

                if phase_I_result["exitflag"] == 1:
                        raise BaseException("Phase I terminated because the␣
↪problem was detected as unbounded.")
```

```
                # Check for feasibility of the initial problem
                if any(phase_I_result["solution"][n:] > np.abs(tol)):
                        result = {
                                "solution" : None,
                                "function" : None,
                                "basis_indices" : None,
                                "iterations" : None,
                                "exitflag" : 3
                        }

                        # Dump some output
                        if verbosity == 'verbose':
                                print('\n\nOriginal problem was found to be␣
↪infeasible in phase I. Phase II won\'t be performed.\n')

                        return result

                # Get initial basis for phase II from solution of phase I
                B = list(set(phase_I_result["basis_indices"]))
                N_real = list(set(range(n)) - set(B))


                # Remove all unwanted artificial indices from basis in case of␣
↪degeneracy (postprocessing)
                try:
                        ### TODO BEGIN ###
                        # Get largest basis index
                        # max_basis_index = ...

                        # If largest basis index is artificial, swap it out
                        while ...:
                                # Obtain representations for nonbasic columns␣
↪of A in current basis

                                # Determine swap index as the largest of the␣
↪ones contributing most to e_max_basis_index (guaranteed to be nonzero)

                                # Swap out largest index in B

                                # Dump some output
                                if verbosity == 'verbose':
                                        print("Replaced artificial basis index␣
↪%4d with real index %4d." % (max_basis_index, entering_basis))

                                # Get new largest basis index
                                # max_basis_index = ...
                        ### TODO END ###
```

```
                except:
                        raise ValueError("Index swap failed in␣
↪phase-I-post-processing.")


        else: # B was not None
                # Dump some output
                if verbosity == 'verbose':
                        print("Initial basis provided.")

        # Run the phase II optimization
        if verbosity == 'verbose':
                print("\nStarting Optimization.")
        result = run_primal_simplex_method(A, b, c, B, entering_rule,␣
↪leaving_rule, parameters)

        return result

def run_primal_simplex_method(A, b, c, B, entering_rule, leaving_rule,␣
↪parameters):
        """
        Solve a linear optimization problem in standard form, i.e., a problem of
        the form
                min c*x
                s.t. Ax = b, x >= 0,
        using the primal simplex method with Bland's rule.
        Expects a Basis for an initial feasible basic point to be supplied.

        Accepts:
                                c: the linear cost's vector
                                A: the constraint's matrix
                                b: the constraint's right-hand-side vector
                                B: basis indices for initial feasible basic vector
                   parameters: optional parameters (dictionary);
                entering_rule: the rule that selects the index that enters the␣
↪basis from the reduced cost vector
                 leaving_rule: the rule that selects the index that leaves the␣
↪basis from the quotient vector
                   parameters: optional parameters (dictionary);
                               the following key/value pairs are evaluated:
                               ["max_iterations"]: maximum number of iterations
                                                ["tol"]: tolerance for optimality␣
↪check on reduced cost vector entries and feasibility in phase I
                                     ["verbosity"]: "verbose" or "quiet"
                                  ["keep_history"]: whether or not to store the␣
↪iteration history (True or False)
```

```
        Returns:
                        result: a dictionary containing the key/value pairs
                                    solution: final iterate
                                    function: the final iterate's objective␣
↪value
                                basis_indices: the final iterate's basis indices
                                   iterations: number of iterations performed
                                    exitflag: flag encoding why the algorithm␣
↪terminated
                                                0: optimal solution found
                                                1: problem is unbounded
                                                2: maximum number of iterations␣
↪reached
        """

        def print_header():
                print('-------------------------------------------------------------------------
                print(' ITER          OBJ      OBJCHNG   MINREDCOST  ␣
↪CHSREDCOST     ENTERING      LEAVING           ')
                print('-------------------------------------------------------------------------

        # Get the algorithmic parameters, using defaults if missing
        max_iterations = parameters.get("max_iterations", 1e3)
        tol = parameters.get("tol", 0)
        verbosity = parameters.get("verbosity", "quiet")
        keep_history = parameters.get("keep_history", False)

        # Define exitflags that will be printed when the algorithm terminates
        exitflag_messages = [
                'Reached optimal solution.',
                'Problem is unbounded.',
                'Reached maximum number of optimization steps.',
        ]

        # Get problem dimensions
        m, n = A.shape

        # Remove all duplicate indices from initial basis and check length
        B = list(set(B))
        if len(B) != m:
                raise ValueError("Size of initial basis does not match problem␣
↪dimension")

        # Initialize initial iterate and non-basis index set from initial basis
        x = np.zeros(n)
        try:
```

```
                x[B] = np.linalg.solve(A[:,B], b)
        except:
                raise ValueError("Could not compute iterate from basis.")
        N = list(set(list(range(n))) - set(B))

        # Check the initial iterate for feasibility
        if any(x < 0):
                raise ValueError("Initial iterate is infeasible.")

        # Initialize variables for simplex iteration loop
        iterations = 0
        exitflag = None
        f_old = np.inf

        # Prepare a dictionary to store the history
        if keep_history:
                history = {
                        "iterates" : [],
                        "objective_values" : [],
                        "basis_indices" : [],
                        "entering_basis" : [],
                        "leaving_basis" : []
                }

        # Perform simplex iterations until termination criterion is met
        while exitflag is None:

                # Compute current function value
                f = c.T @ x

                # Record the current iterate, its function value and the basis
                if keep_history:
                        history["iterates"].append(x.copy())
                        history["objective_values"].append(f)
                        history["basis_indices"].append(B)

                # Dump some output
                if verbosity == 'verbose':
                        if (iterations%10 == 0): print_header()
                        print(' %4d  %11.4e  %11.4e' % (iterations, f,␣
↪f-f_old), end = '')

                # Stop when the maximum number of iterations has been reached
                if iterations >= max_iterations:
                        exitflag = 2
                        break
```

```
                try:
                        ### TODO BEGIN ###
                        # Compute reduced cost vector (sN)
                        reduced_cost = ...
                        ### TODO END ###
                except:
                        raise ValueError("Reduced cost vector could not be␣
↪computed.")


                # Dump some output
                if verbosity == 'verbose':
                        print('  %11.4e' % (np.amin(reduced_cost)), end = '')

                # Check for optimality by checking sign of reduced cost vector␣
↪entries
                # and also check if numerical inaccuracy is prohibiting proper␣
↪termination
                ### TODO BEGIN ###
                if ...:
                ### TODO END ###
                        exitflag = 0
                        break

                # Remember function value
                f_old = f

                ### TODO BEGIN ###
                # Select new entering basis index using Bland's rule
                entering_basis = ...
                ### TODO END ###

                # Dump some output
                if verbosity == 'verbose':
                        print('  %11.4e' % (reduced_cost[entering_basis]), end␣
↪= '')

                # Record the index that enters_basis
                if keep_history: history["entering_basis"].
↪append(entering_basis)

                # Dump some output
                if verbosity == 'verbose': print('         %4d  ' %␣
↪(entering_basis), end = '')

                ### TODO BEGIN ###
```

```
                # Compute update direction
                delta_x = ...
                ### TODO END ###

                ### TODO BEGIN ###
                # Check for unboundedness
                if ...:
                        exitflag = 1
                        break
                ### TODO END ###

                ### TODO BEGIN ###
                # Compute steplength from quotient test
                quotients = ...
                step_length = ...
                ### TODO END ###

                ### TODO BEGIN ###
                # Compute index that will leave the basis
                leaving_basis = ...
                ### TODO END ###

                # Record the leaving basis index
                if keep_history: history["leaving_basis"].append(leaving_basis)

                # Dump some output
                if verbosity == 'verbose': print('       %4d   ' %␣
↪(leaving_basis))

                # Update iterate
                ### TODO BEGIN ###
                x ...
                ### TODO END ###

                # Update basis and nonbasis
                ### TODO BEGIN ###
                B = ...
                N = ...
                ### TODO END ###

                iterations = iterations + 1

        # Dump some output
        if verbosity == 'verbose':
                print('\n\nThe simplex method is exiting with flag %d.\n'␣
↪%(exitflag) + str(exitflag_messages[exitflag])+'\n' )
```

```
        # Create and populate the result to be returned
        result = {
                "solution" : x,
                "function" : c.T @ x,
                "basis_indices" : B,
                "iterations" : iterations,
                "exitflag" : exitflag
        }

        # Assign the history to the result if required
        if keep_history: result["history"] = history

        return result
```

1.3 Test der Simplex-Implementierung

Die Implementierung des Simplex hat einige Fallunterscheidungen anzustellen. Um zu überprüfen, ob Ihre Implementierung (weitestgehend) korrekt ist, stellen wir Ihnen in der nächsten Zelle ein paar Testfälle zur Verfügung, anhand derer sie ihre Implementierung testen können.

Aufgabe: Führen Sie die unten Zelle aus, und prüfen Sie, ob ihr Simplex die Testfälle lösen kann.

[ ]:
```
# This script tests the simplex method for various scenarious for debugging␣
 ↪purposes
# The following tests are performed using bland's rule for entering and exiting:
# 1. Can the simplex method detect an infeasible initial basis correctly␣
 ↪(Mozart problem)
# 2. Can the Mozart problem be solved correctly from given initial basis
# 3. Can the Mozart problem be solved correctly with a phase I search␣
 ↪beforehand to compute initial basis
# 4. Check if Phase I can detect infeasibility of the problem correctly␣
 ↪(infeasibly modified Mozart problem)
# 5. Check if a degenerate basis returned from Phase I is modified to be␣
 ↪feasible  for phase II in post-processing

# Not checked: Detection of unboundedness of the initial problem

import numpy as np

# Set parameters for simplex method
simplex_parameters = {
        "max_iterations" : 1000,
        "verbosity" : "verbose",
        "keep_history" : True
}

# Define the mozart problem
```

```
A_mozart = np.array([[1, 1, 1, 0, 0],
                     [2, 1, 0, 1, 0],
                     [1, 2, 0, 0, 1]])
c_mozart = np.array([-9, -8, 0, 0, 0])
b_mozart = np.array([6, 11, 9])

# Modify the Mozart problem to become infeasible by fixing x_1 to -1
A_mozart_infeasible = np.vstack((A_mozart, [1, 0, 0, 0, 0]))
b_mozart_infeasible = np.hstack((b_mozart, [-1]))

# Define a problem that requires post processing of phase I result
n = 10
m = 8
A_post_processing = -np.hstack((np.eye(m),np.ones((m,n-m))))
b_post_processing = np.zeros(m)
c_post_processing = np.ones(n)

# Start checking
# Check if infeasible initial point is detected
print('###### Check 1: Checking if infeasible initial point is detected')
try:
        result = primal_simplex_method(A_mozart, b_mozart, c_mozart, B = [0, 3,␣
 ↪4], parameters = simplex_parameters)
except ValueError as err:
        print("Simplex error: {0}\n".format(err))
        if(str(err) == "Initial iterate is infeasible."):
                print("-----> Check 1 passed")
        else:
                raise BaseException("Infeasible initial basis not handled␣
 ↪correctly")

# Check if mozart problem can be solved correctly
print('###### Check 2: Checking if Mozart problem can be solved correctly from␣
 ↪feasible point')
result = primal_simplex_method(A_mozart, b_mozart, c_mozart, B = [0, 2, 4],␣
 ↪parameters = simplex_parameters)
if result["exitflag"] == 0 and set(result["basis_indices"]) == {0, 1, 4}:
        print("-----> Check 2 passed")
else:
        raise BaseException("Mozart problem not solved correctly")

# Check if phase I works without degeneracy in the phase I solution basis
print('###### Check 3: Checking if phase I works without degeneracy in the␣
 ↪phase I solution basis')
result = primal_simplex_method(A_mozart, b_mozart, c_mozart, B = None,␣
 ↪parameters = simplex_parameters)
```

```
if result["exitflag"] == 0 and set(result["basis_indices"]) == {0, 1, 4}:
        print("-----> Check 3 passed")
else:
        raise BaseException("Mozart problem not solved correctly including␣
 ↪phase I")

# Check detecting of infeasibility in phase I
print('###### Check 4: Checking if infeasibility in problem is detected')
result = primal_simplex_method(A_mozart_infeasible, b_mozart_infeasible,␣
 ↪c_mozart, B = None, parameters = simplex_parameters)
if result["exitflag"] == 3:
        print("-----> Check 4 passed")
else:
        raise BaseException("Infeasiblity not handled correctly")

# Check if degenerate basis in phase I is handled correctly
print('###### Check 5: Checking if degenerate phase I basis is handled␣
 ↪correctly')
result = primal_simplex_method(A_post_processing, b_post_processing,␣
 ↪c_post_processing, B = None, parameters = simplex_parameters)
if all(np.array(result["basis_indices"]) < n):
        print("-----> Check 5 passed")
else:
        raise BaseException("Degenerate phase I basis not handled correctly")
```

1.4 Einfluss der Index-Auswahlregeln

Die obigen Tests wurden mit den Auswahlregeln von Bland durchgeführt, die in der Veranstaltung besprochen wurde. Wir wollen jetzt an einem ausgewählten Beispiel das Verhalten der beiden Verfahren gegenüberstellen.

Aufgabe: Lösen Sie das unten stehende Problem mit Ihrem Simplex Verfahren – einmal für die Index-Auswahlverfahren-Kombination Steepest-Descent für *entering* und der Regel von Bland für *leaving* sowie einmal für die Regel von Bland (sowohl für *entering* und *leaving*) – indem Sie die nächste Zelle ausführen.

[ ]:
```
# Set parameters for the simplex method
simplex_parameters = {
        "max_iterations" : 20,
        "verbosity" : "verbose",
        "keep_history" : True
}

# Set the data
A = np.array([[0.25, -8, -1,   9, 1, 0, 0],
              [0.5, -12, -0.5, 3, 0, 1, 0],
              [0,     0,  1,   0, 0, 0, 1]])
```

```
c = np.array([-0.75, 20, -0.5, 6, 0 ,0, 0])
b = np.array([0, 0, 1])

# Solve the problem using steepest descent rule in simplex algorithm
result_steepest_descent = primal_simplex_method(A, b, c, B = [4, 5, 6],␣
 ↪entering_rule = steepest_descent_entering_rule, parameters =␣
 ↪simplex_parameters)

# Solve the problem using bland
result_bland = primal_simplex_method(A, b, c, B = [4, 5, 6], parameters =␣
 ↪simplex_parameters)
```

Aufgabe: Beschreiben Sie Ihre Beobachtung zum Verhalten des Algorithmus in beiden Fällen. Inwiefern entspricht das Ihrer Erwartung?

TODO Ihre Antwort hier

1.5 Toleranzen in der Implementierung

Bisher haben wir alle Berechnungen mit Algorithmusparametern durchgeführt, die keine Toleranzen berücksichtigt haben. Das war bei den obigen Problemen kein Problem, im Allgemeinen muss das aber keine gute Idee sein, wie wir hier sehen werden.

Aufgabe: Führen Sie die nachfolgende Zelle aus.

[ ]:
```
# This script visualizes the degenerate behavior of the simplex algorithm when␣
 ↪tolerances are set to 0.
# We solve a number of random examples until we find one where we expect␣
 ↪numerical issues and rerun that
# example again in verbose mode to analyze the output
import sys
sys.path.append('src/')
from visualization_functions import *

# Set parameters for simplex method
simplex_parameters = {
        "max_iterations" : 150,
        "verbosity" : "quiet",
        "keep_history" : False
}

# Set parameters for random testing
max_n = 50
min_n = 5
max_random_entry = 10

# Create pseudo random number generator with reproducable seed
np.random.seed(0)
```

```
rng = np.random.default_rng(np.random.MT19937(seed=0))

#Get pseudo random data sizes from sampling
number_of_samples = 1000
N = rng.integers(min_n,max_n+1,number_of_samples)
M = rng.integers(1, N, number_of_samples)

# Solve test instances until we find one that needs max iterations
for n, m in zip(N, M):
        # Dump some output
        print('########################## Trying out n = %d, m = %d␣
 ↪##########################' % (n,m))

        # Create pseudo random optimizer data (primal and dual)
        x_opt = np.hstack((np.zeros(m), max_random_entry * rng.random(n-m)))
        s_opt = np.hstack((max_random_entry * rng.random(m), np.zeros(n-m)))
        lam = max_random_entry * (rng.random(m) - 0.5)

        # Create pseudo random constraint matrix
        A = np.hstack((np.eye(m), rng.random((m, n-m))))

        # Create constraint rhs
        b = A @ x_opt

        # Create cost functional for which x_opt is an optimizer
        c = A.T @ lam + s_opt

        # Create basis
        B = np.arange(0, m)

        # Compute optimal function value
        f_opt = c @ x_opt

        # Start simplex iterations and store results plus problem dimensions
        result = primal_simplex_method(A, b, c, B, parameters =␣
 ↪simplex_parameters)

        if result["exitflag"] == 2:
                print('Maximum iterations needed. Running again in verbose mode.
 ↪')
                break;

# Rerun the example in verbose mode
simplex_parameters["verbosity"] = 'verbose'
result = primal_simplex_method(A, b, c, B, parameters = simplex_parameters)
```

Aufgabe: Beschreiben Sie das beobachtete Verhalten und geben Sie eine Begründung für das

Verhalten.

TODO Ihre Antwort hier

1.6 Simplex-Performance

Wir wissen aus dem Skript bereits, dass das Simplex-Verfahren zwar meist "schnell" doch manchmal "langsam" ist. Konkreter ist der worst-case Zusammenhang der Simplex-Iterationen und der Raumdimension midnestens exponentiell. Das Beispiel, dass das zeigt ist der *Klee-Minty Würfel* mit einem dazugehörigen Kostenvektor. Eine Variante des Problems ist das Folgende:

$$
\begin{array}{rll}
\text{Maximiere} & 2^{n-1}x_1 + 2^{n-2}x_2 + \ldots + 2x_{n-1} + x_n & \text{über } x \in \mathbb{R}^n \\
& x_1 \qquad\qquad\qquad\qquad \leq\ 5 & \\
& 4x_1 + \quad x_2 \qquad\qquad \leq\ 25 & \\
\text{sodass} & 8x_1 + \quad 4x_2 + x_3 \qquad \leq 125 & \\
& \vdots \qquad\qquad\qquad\qquad\ \vdots & \\
& 2^n x_1 + 2^{n-1}x_2 + \ldots + 4x_{n-1} + x_n \leq\ 5^n & \\
\text{und} & x \geq 0. &
\end{array}
$$

Aufgabe: Vervollständigen Sie den Code in der folgenden Zelle um das Problem von Klee-Minty in so vielen wachsenden Raumdimensionen zu lösen, wie es Ihr Rechner in vernünftiger Zeit hergibt (die Laufzeit steigt exponentiell! Viele werden es nicht sein) und führen Sie die Zelle aus.

[ ]:
```
# This script illustrates the simplex algorithm's bad case complexity on a␣
 ↪Klee-Minty cube

from visualization_functions import *

# Set maximum cube dimension (actual spatial dimension, not problem size n in␣
 ↪normal form)
cube_max_dim = 10

# Set parameters for simplex method
simplex_parameters = {
        "max_iterations" : 2**cube_max_dim,
        "tol" : 1e-15,
        "verbosity" : "quiet",
        "keep_history" : True
}

# Initialize container for the iterations needed
iterations = []
dimensions = range(2, cube_max_dim+1)

for n in dimensions:
        ### TODO BEGIN ###
```

```
        # Set the data for the cube
        A = ...
        b = ...

        # Set the cost functional vector
        c = ...
        ### TODO END ###

        # Set initial basis
        initial_basis = list(range(n, 2*n))

        # Solve the problem using steepest descent
        result = primal_simplex_method(A, b, c, B = initial_basis,␣
 ↪entering_rule = steepest_descent_entering_rule, parameters =␣
 ↪simplex_parameters)
        # Remember iteration numbers
        iterations.append(result["iterations"])

        # Plot cube if plottable
        if n <= 3:
                plot_simplex_iterates(np.vstack([A[:,:n], -np.eye(n)]), np.
 ↪hstack([b, np.zeros(n)]), result["history"]["iterates"], np.ones(n))
                if n == 3:
                        plt.gca().view_init(elev=10., azim=20)


# Output required iterations
for n in range(2, cube_max_dim+1):
        print("Iterations required to solve problem with cube dimension %4d:␣
 ↪%d" % (n, iterations[n-2]))

# Plot iteration numbers and the exponential
plot_klee_minty_iteration_numbers(dimensions, iterations)
```

Dass dieser “bad-case” nicht repräsentativ für die “typische” Simplex-Performance ist, wollen wir uns anhand der folgenden, randomisierten Beispiele veranschaulichen.

Aufgabe: Führen Sie die folgende Zelle aus.

[ ]:
```
# This script visualizes the performance of the simplex algorithm for various␣
 ↪randomly generated data sets.
# The "random" data sets are generated as follows
#
# Set n and set 0 < m < n
# Set x* with m random positive numbers in random positions, other components␣
 ↪are zero
```

```
# Set s* with n-m random positive numbers and zeros in complementary positions␣
 ↪to x^*
# Set A as a mxn matrix with random numbers
# Set b* = Ax*
# Set lam_optbda* as random vector
# Set c = A^T lam_optbda* + s*
#
# This way, the first order optimality conditions are satisfied (which are also␣
 ↪sufficient for LPs), i.e.,
# x* is a solution to the LP.
# Note that x* will also be a vertex (a feasible basic vector) - see the number␣
 ↪of zero entries.

# Note that generally, A can not be guaranteed to have full rank.

import numpy as np
from visualization_functions import *

# Set parameters for simplex method
simplex_parameters = {
        "max_iterations" : 1000,
        "tol" : 1e-15,
        "verbosity" : "quiet",
        "keep_history" : True
}

# Set parameters for random testing
max_n = 100
min_n = 5
max_random_entry = 100
results = []

# Create pseudo random number generator
rng = np.random.default_rng(np.random.MT19937())

#Get pseudo random data sizes from sampling
number_of_samples = 500
N = rng.integers(min_n,max_n+1,number_of_samples)
M = rng.integers(1, N, number_of_samples)

# Solve test instance for all samples
for n, m in zip(N, M):
        # Dump some output
        print('########################## n = %d, m = %d␣
 ↪##########################' % (n,m))

        # Create random x_opt with m positive nonzero entries
```

```
        x_opt = max_random_entry * rng.random(n)
        x_opt[rng.choice(n, n-m, replace = False)] = 0

        # Create random s^* with n-m positive nonzero entries complementary to␣
↪x_opt
        s_opt = max_random_entry * rng.random(n)
        s_opt = np.where(x_opt > 0, 0, s_opt)

        # Create random lambda_opts
        lam_opt = max_random_entry * (rng.random(m) - 0.5)

        # Create pseudo random constraint matrix
        A = max_random_entry * (rng.random((m, n)) - 0.5)

        # Create constraint rhs
        b = A @ x_opt

        # Create cost functional for which x_opt is an optimizer
        c = A.T @ lam_opt + s_opt

        # Compute optimal function value
        f_opt = c @ x_opt

        # Start simplex iterations and store results plus problem dimensions
        results.append(primal_simplex_method(A, b, c, B = None, parameters =␣
↪simplex_parameters))
        results[-1]["n"] = n
        results[-1]["m"] = m
        results[-1]["norm_of_distance"] = np.linalg.
↪norm(results[-1]["solution"]-x_opt)
        results[-1]["diff_in_objective"] = results[-1]["function"]-f_opt

        # Dump some output
        if results[-1]["exitflag"] == 0:
          print('Solution found.')
        else:
          print('No solution found.')

        print('Norm of distance between x_opt and solution is: %11.4e' %␣
↪results[-1]["norm_of_distance"])
        print('Difference in function values between f and f_opt is: %11.4e' %␣
↪results[-1]["diff_in_objective"])

# Plot iterations over n
plot_simplex_iterations_over_dimension(results)

# Plot iterations over sum of n and m
```

```
plot_simplex_iterations_over_sum(results)

# Plot iterations over quotient of m and n
plot_simplex_iterations_over_quotient(results)
```

Aufgabe: Interpretieren Sie die Ergebnisse.

TODO Ihre Antwort hier

1.7 Berechnung von Tschebyschow-Zentren

Zuletzt wollen wir noch zurück zu der Übungsaufgabe der *innersten Punkte* in Polyedern - den Tschebyschow Zentren. Wir wollen einige davon Berechnen und uns die Lösungseigenschaften etwas genauer ansehen.

Aufgabe: Vervollständigen Sie den Code in der folgenden Zelle, in dem Sie mindestens ein Polyeder in je zwei und drei Raumdimensionen konstruieren, dessen Tschebyschow Zentrum berechnen und darstellen.

[ ]:
```
# This module computes the Tschebyschow center for various polyhedra

from visualization_functions import *

def compute_and_plot_tschebyschow_center(halfspace_normals, halfspace_values,␣
↪initial_bases):
        """ Compute and visualize Tschebyschow center for each polyhedron in␣
↪list"""
        for normals, values, initial_basis in zip(halfspace_normals,␣
↪halfspace_values, initial_bases):
                # Get problem dimensions
                m, n = normals.shape

                # Construct data for standard form of Tschebyschow center␣
↪problem
                A = np.hstack([
                        normals,
                        -normals,
                        np.eye(normals.shape[0]),
                        np.array([np.linalg.norm(normals, ord = 2, axis = 1)]).
↪T])
                b = values
                c = np.zeros(A.shape[1])
                c[-1] = -1

                # Set parameters for simplex method
                simplex_parameters = {
                        "max_iterations" : 1000,
                        "tol" : 1e-15,
```

```
                        "verbosity" : "quiet",
                        "keep_history" : True
                    }

                    # Solve for Tschebyschow center and radius
                    result = primal_simplex_method(A, b, c, B = initial_basis,␣
 ↪parameters = simplex_parameters)

                    # Reconstruct solution
                    sol = result["solution"]
                    center = sol[0:n] - sol[n:2*n]
                    radius = sol[-1]

                    plot_tschebyschow_center(normals, values, center, radius)


# Create some empty containers
halfspace_normals = []
halfspace_values = []
initial_bases = []

# Create data for a 2d polyhedron
### TODO BEGIN ###
halfspace_normals.append(...)
halfspace_values.append(...)
### TODO END ###
initial_bases.append(None)

# Create data for a 3 d polyhedron
### TODO BEGIN ###
halfspace_normals.append(...)
halfspace_values.append(...)
### TODO END ###
initial_bases.append(None)

compute_and_plot_tschebyschow_center(halfspace_normals, halfspace_values,␣
 ↪initial_bases)
```

Wir wollen uns nun noch ansehen, welches Verständnis von “Ecken” wir in der Aufgabe der Berechnung von Tschebyschow-Zentren wir entwickeln können.

Aufgabe: Berechnen und plotten Sie Lösungen der Tschebyschow-Zentrums-Aufgaben für die von uns vorgegebenen Polyeder indem sie die Folgende Zelle ausführen.

[ ]:
```
# Create some empty containers
halfspace_normals = []
halfspace_values = []
initial_bases = []
```

```
# Create a rectangular polyhedron for "vertex" comparison
halfspace_normals.append(np.array([[-1,   0],
                                   [ 0,  -1],
                                   [ 1,   0],
                                   [ 0,   1]]))
halfspace_values.append(np.array([0, 0, 1, 2]))
initial_bases.append(np.arange(4,8))

# Create a rectangular polyhedron for "vertex" comparison
halfspace_normals.append(np.array([[-1,   0],
                                   [ 0,  -1],
                                   [ 1,   0],
                                   [ 0,   1]]))
halfspace_values.append(np.array([0, 0, 2, 1]))
initial_bases.append(np.arange(4,8))

# Create a special rectangular polyhedron for "vertex" comparison
halfspace_normals.append(np.array([[-1,   0],
                                   [ 0,  -1],
                                   [ 1,   0],
                                   [ 0,   1]]))
halfspace_values.append(np.array([2, 1, 3, 1]))
initial_bases.append(np.arange(4,8))

compute_and_plot_tschebyschow_center(halfspace_normals, halfspace_values,␣
 ↪initial_bases)
```

Aufgabe: Erklären Sie, warum die Lösung im letzten Fall nur 2, statt 3 Hyperebenen berührt. Kann es auch einen optimalen Basisvektor in den ersten zwei Fällen geben, der nur zwei Hyperebenen berührt? Erklären Sie den Zusammenhang von Basislösungen in dem Problem der Tschebyschow-Zentren und der Anzahl der Hyperebenen, die berührt werden.

TODO Ihre Antwort hier