THE UNIVERSITY of York

CONVEX OPTIMIZATION
theory and practice

Constantinos Skarakis

BT Group plc
Mobility Research Centre,
Complexity Research
Adastral Park
Martlesham Heath
Suffolk IP5 3RE
United Kingdom

University of York
Department of Mathematics,
Heslington
York, YO10 5DD
United Kingdom

Supervisors: Dr. K. M. Briggs
Dr. J. Levesley

Dissertation submitted for the MSc in Mathematics with Modern Applications, Department of Mathematics, University of York, UK on August 22, 2008

Contents

Chapter 1

Introduction

The purpose of this paper is to bring closer the mathematical and the computer science, in the field where it makes the most sense to do so: **Mathematical optimization**. More specifically **Convex optimization**. The reason is simply because most real problems in engineering, information technology, or many other industrial sectors involve optimizing some kind of variable. That can either be minimizing cost, emissions, energy usage or maximizing profit, radio coverage, productivity. And most of the times, if not always, these problems are either difficult, or very inefficient to solve by hand. Thus the use of technology is required.

The advantages of convex optimization, is that first of all, it includes a large number of problem classes. In other words, many of the most commonly addressed optimization problems are convex. Or even if they are not convex, some times [sections 3.1, 3.3.1], there is a way to reformulate them into convex form. What's even more important than that, is that due to the convex nature of the feasible set of the problem, any local optimum is also the global optimum. Algorithms written to solve convex optimization problems take advantage of such particularities, and are faster, more efficient and very reliable. Especially in specific classes of convex programming, such as linear or geometric programing, algorithms have been written to deal with these particular cases even more efficiently. As a result, very large problems, with even thousands of variables and constraints, are solvable in sufficiently little time.

The event that inspired this project, was the relatively recent release of **CVXMOD**; a programming tool for solving convex optimization problems, written in the programming language **python**. The release of an optimization software is not something new. Similar tasks can be performed using Matlab solvers, like **cvx**, but it normally takes an above medium level of expertise and many lines of code and maybe also include signifi-

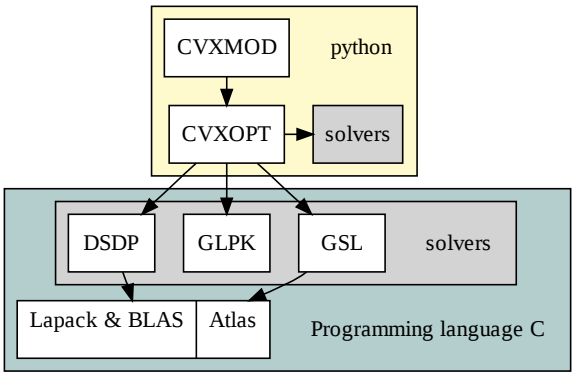

Figure 1.1: **CVXMOD**, **CVXOPT** and their dependencies.

cant cost from the purchase of the program. What's really exciting about this new piece of software, is that it is free, open-source, and it makes it possible to solve a large complex convex optimization problem using just 15 or 20 lines of code. It works as a modeling layer for **CVXOPT**, a python module using the same principles with **cvx**.

CVXOPT comes with its own solvers written completely in python. However, the user has the option to use in some cases C solvers, translated into **python**. These are the same solvers used by **cvx** in Matlab, and rely on the same C libraries [Frigo and Johnson, 2003, Makhorin, 2003, Benson and Ye, 2008, Galassi et al., 2003, Whaley and Petitet, 2005]. These libraries have been used for many years and as a result they have been well tested and constitute to a very reliable set of optimization tools. Among other things, we hope to test the efficiency of **CVXMOD** and **CVXOPT** in our examples. If the results are satisfactory, that means that we will have in our hands tools that are very high-level, as the programming language they are written in themselves, as well as a good mathematical tool. Then the only difficulty will lie in formulating the problem mathematically. Here is a list of applications we present as example problems that can be efficiently solved using **CVXOPT** [Boyd and Vandenberghe, 2004]

- Optimal trade-off curve for a regularized least-squares problem.
- Optimal risk-return trade-off for portfolio optimization problems.
- Robust regression.
- Stochastic and worst-case robust approximation.

```
from cvxmod import *
c=matrix((−1,1),(2,1))
A=matrix((−1.5,0.25,3.0,0.0,−0.5,1.0,1.0,−1.0,−1.0,−1.0),(5,2))
b=matrix((2.0,9.0,17.0,−4.0,−6.0),(5,1))
x=optvar('x',2)
p=problem(minimize(tp(c)*x),[A*x<=b])
p.solve()
```

Program 1.1: *lpsimple.py*

- Polynomial and spline fitting.
- Least-squares fit of a convex function.
- Logistic regression.
- Maximum entropy distribution with constraints.
- Chebyshev and Chernoff bounds on probability of correct detection.
- Ellipsoidal approximations of a polyhedron.
- Centers of a polyhedron.
- Linear discrimination.
- Linear, quadratic and fourth-order placement.
- Floor planning.
- Optimal hybrid vehicle operation.
- Truss design [Freund, 2004].

At this point, we present a very simple example. We will use **CVXMOD** to solve the following linear program:

$$\begin{array}{ll} \text{minimize} & y - x, \\ \text{subject to} & y \leqslant 1.5x + 2, \\ & y \leqslant -0.25x + 9, \\ & y \geqslant 3.0x - 17, \\ & y \geqslant 4, \\ & y \geqslant -0.5x + 6. \end{array}$$

We arrange the coefficient of the objective and constraints appropriately

$$c = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \qquad A = \begin{bmatrix} -1.5 & 1 \\ 0.25 & 1 \\ 3 & -1 \\ 0 & -1 \\ -0.5 & -1 \end{bmatrix}, \qquad b = \begin{bmatrix} 2 \\ 9 \\ 17 \\ -4 \\ -6 \end{bmatrix},$$

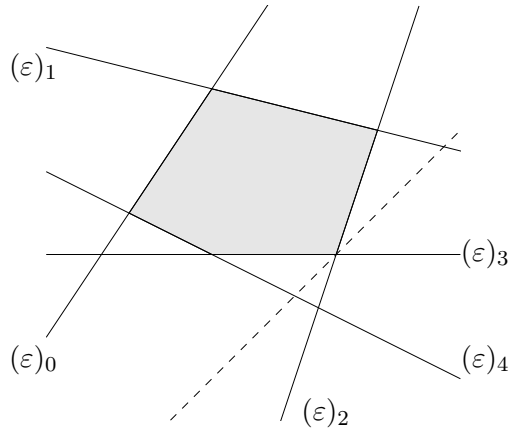

Figure 1.2: $(\varepsilon)_0 : y = 3x/2 + 2$, $(\varepsilon)_1 : y = -x/4 + 9$, $(\varepsilon)_2 : y = 3x - 17$, $(\varepsilon)_3 : y = 4$, $(\varepsilon)_4 : y = -x/2 + 6$

so we can rewrite the problem using a more efficient description. Now the objective can be written as $c^T x$ and all five constraints are included in $Ax \preccurlyeq b$.

Program 1.1 will provide the optimal solution, which is -3. In Figure 1.2, one can get a better visualization of problem. The shaded area is the feasible region of the problem, determined by the constraints. We seek the line $y = x + \alpha$, that intersects the feasible set for the minimum value of α. The dashed line, is the optimal solution we are looking for. Since the program gave output -3, the dashed line will be

$$y = x - 3.$$

Chapter 2

Theoretical Background

The definition and results given below are mostly based on and in accordance with Boyd and Vandenberghe [2004].

For the sake of a better visual result, and also to maintain consistency between the mathematical formulation and the code implementation of applications, we will be using notation somewhat outside the pure mathematical formalism. First of all, we will enumerate starting from zero, which means that all vector and matrix indices will start from zero and so will summations and products. Also, we will use the symbol '$\widehat{\ }$' for "probability" quantities, to denote complement with respect to 1. In other words, if $\lambda \in [0, 1]$ then $\widehat{\lambda} = 1 - \lambda$.

2.1 Convex sets

A (geometrical) set C is *convex*, when for every two points that lie in C, every point on the line segment that connects them, also lies in C.

Definition 1. Let $C \subseteq \mathbb{R}^n$. Then C is *convex*, if and only if for any $x_0, x_1 \in C$ and any $\lambda \in [0, 1]$

$$\widehat{\lambda} x_0 + \lambda x_1 \in C. \tag{2.1}$$

The above definitions can be generalised inductively for more than two points in the set. So we can say that a set is convex, if and only if it contains all possible *weighted averages* of its points, which means all linear combinations of its points, with coefficients that sum up to 1. Such combinations are also called *convex combinations*. The set

$$\{\lambda_0 x_0 + \cdots + \lambda_{k-1} x_{k-1} \mid x_i \in S, i = 0 \ldots k-1, \lambda_i \geqslant 0, \sum_{0 \leqslant i < k} \lambda_i = 1\}, \tag{2.2}$$

is the set of all possible convex combinations of points of a set S and is called the *convex hull* of S. The convex hull is the smallest convex set that includes S. It could also be described as the intersection of all convex sets that include S. Convex sets are very common in geometry and very important in numerous mathematical disciplines.

Affine sets

These are the most immediate, or even trivial example of convex sets.

Definition 2. A set $A \subseteq \mathbb{R}^n$ is affine, if for any $x_0, x_1 \in A$ and $\lambda \in \mathbb{R}$

$$\widehat{\lambda} x_0 + \lambda x_1 \in A. \tag{2.3}$$

What the definition formally states is that a set A is affine, if for any two points that lie in A, every point on the line through these two points, also lies in A. The simple example of an affine set, is of course the straight line. Affine sets are convex. However the converse is not true. Convex sets are not (in general) affine.

Hyperplanes

Definition 3. A hyperplane is a set of the form

$$\{x \mid a^T x = b\}, \tag{2.4}$$

where $x, a \in \mathbb{R}^n$, $b \in \mathbb{R}$ and $a \neq \mathbf{0}$.

Geometrically speaking, a *hyperplane* is a translation of a subspace of $\mathbb{R}^n$ of dimension $n-1$. Most importantly, its elements, as elements of $\mathbb{R}^n$, are linearly dependent. For example, the hyperplanes of $\mathbb{R}^2$ will be the straight lines. In $\mathbb{R}^4$, the hyperplanes will be 3-dimensional spaces. In $\mathbb{R}^3$ hyperplanes will be nothing more than normal planes. For that reason, from now forth we will use the term *plane* instead of hyperplane.

Definition 4. A (closed) halfspace, is a set of the form

$$\{x \mid a^T x \leqslant b\}, \tag{2.5}$$

where $x, a \in \mathbb{R}^n$, $b \in \mathbb{R}$ and $a \neq \mathbf{0}$. If the inequality is strict, then we have an *open* halfspace.

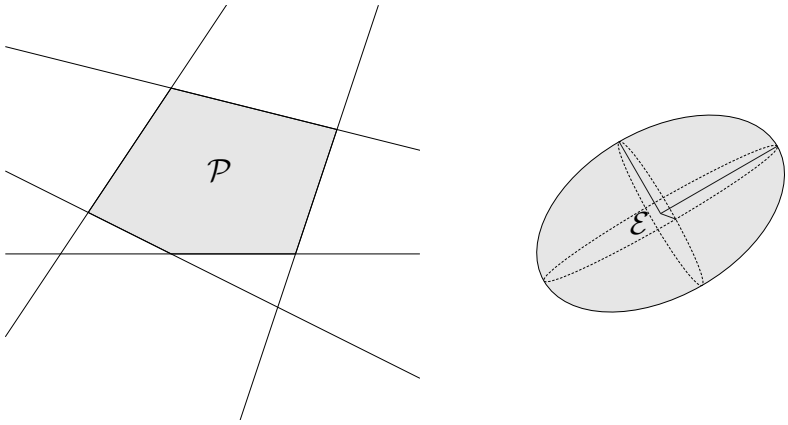

Figure 2.1: Left: A polyhedron in two dimensions, represented as the intersection of 5 halfspaces. Right: A three dimensional ellipsoid.

A plane divides the space in two halfspaces. For example the plane

$$\{x \mid a^T x = b\},$$

will divide $\mathbb{R}^n$ into the halfspaces $\{x \mid a^T x < b\}$ and $\{x \mid -a^T x < -b\}$.

Since planes are collections of linearly dependent points, they are affine and consequently, they are convex. On the other hand, halfspaces are convex sets, but they are not affine, because they only include lines that are parallel to the boundary-plane.

Polyhedra

An intersection of halfspaces and hyperplanes is called a *polyhedron*. If a polyhedron is bounded, it can be called a *polytope*.

Definition 5. A polyhedron is a set of the form

$$\mathcal{P} = \{x \mid a_i^T x \leqslant b_i, i = 0, \ldots, m-1,\ c_j x = d_j, j = 0, \ldots, p-1\}. \tag{2.6}$$

The most immediate nontrivial example of a polyhedron in two dimensions, would a polygon, or just the interior of the polygon, if we demand strict inequalities in the definition. Needless to say, that if the system of equalities and inequalities in (2.6) have no solution, then the $\mathcal{P}$ will be the empty set.

Polyhedra are convex sets. A proof of this statement, would be a special case to the proof that intersections of convex sets are convex:

Lemma 1. *Any countable intersection of convex sets, is a convex set.*

Proof. We will prove the statement for the intersection of two sets. The rest of the proof is simple induction. Let C, D be convex subsets of $\mathbb{R}^n$. Let $x_0, x_1 \in C \cap D$ and let $\lambda \in [0,1]$

$$C \cap D \subseteq C \Rightarrow x_0, x_1 \in C \Rightarrow$$
$$\widehat{\lambda} x_0 + \lambda x_1 \in C, \tag{2.7}$$
$$C \cap D \subseteq D \Rightarrow x_0, x_1 \in D \Rightarrow$$
$$\widehat{\lambda} x_0 + \lambda x_1 \in D. \tag{2.8}$$

(2.7) and (2.8)$\Rightarrow$

$$\widehat{\lambda} x_0 + \lambda x_1 \in C \cap D. \qquad \square$$

A very important polyhedron in the field of mathematical optimization is the *simplex*.

Definition 6. Let $v_0, v_1, \dots v_k \in \mathbb{R}^n$, $k \leqslant n$ such that $v_1 - v_0$, $v_2 - v_0$, $\dots v_k - v_0$ are linearly independent. A simplex, is a set of the form

$$\{\theta_0 v_0 + \cdots + \theta_k v_k \mid \theta_j \geqslant 0, j = 0, \dots, k-1, \sum_{0 \leqslant j < k} \theta_j = 1\}. \tag{2.9}$$

In two dimension, an example of a simplex can be any triangle and in three dimensions any tetrahedron.

Norm Balls

A *norm ball* - or just *ball*, if the norm we are referring to is the Euclidean - is well known to be a set of the form

$$\{x \in \mathbb{R}^n \mid \|x - x_c\| \leqslant r\}. \tag{2.10}$$

Point x_c is the center of the norm ball, $r \in \mathbb{R}$ is its radius and $\|\cdot\|$ is any norm on $\mathbb{R}^n$.

It is easy to prove convexity of norm balls.

Proof. Let B be a norm ball, in other words, a set of the form (2.10). Let $\lambda \in [0,1]$.

$$x_0, x_1 \in B \Rightarrow \|x_0 - x_c\| \leqslant r,\ \|x_1 - x_c\| \leqslant r$$
$$\|\widehat{\lambda} x_0 + \lambda x_1 - x_c\| = \|\widehat{\lambda} x_0 + \lambda x_1 - \widehat{\lambda} x_c - \lambda x_c\| \leqslant$$
$$\widehat{\lambda}\|x_0 - x_c\| + \lambda\|x_1 - x_c\| \leqslant \widehat{\lambda} r + \lambda r = r. \qquad \square$$

Ellipsoids

A generalization of the 2 dimensional conic section, the *ellipse*, is the *ellipsoid*. In 3 dimensions, it's what one might describe as a melon-shaped set. A rigorous definition would be

Definition 7. An ellipsoid is a set of the form

$$\{x_c + Au \mid \|u\|_2 \leqslant 1\}, \tag{2.11}$$

where $x_c \in \mathbb{R}^n$ is the center of the ellipsoid, $A \in \mathbb{R}^{n\times n}$ is a square matrix and $\|\cdot\|_2$ is the Euclidean norm. If A is singular, the set is called a *degenerate* ellipsoid.

A more practical definition of an ellipsoid will be given after definiteness has been defined. If A was replaced by a real number r, then (2.11) would be an alternative representation of the Euclidean ball. Ellipsoids are also convex:

Proof. Let $\mathcal{E}$ be a set of the form (2.11). Let $\lambda \in [0,1]$ and $u_0, u_1 \in \mathbb{R}^n$ with $\|u_0\|_2, \|u_1\|_2 \leqslant 1$.

$$x_c + Au_0, x_c + Au_1 \in \mathcal{E} \Rightarrow$$

$$\widehat{\lambda}x_c + \widehat{\lambda}Au_0 + \lambda x_c + \lambda Au_1 = x_c + A\big(\widehat{\lambda}u_0 + \lambda u_1\big) \in \mathcal{E},$$

because

$$\|\widehat{\lambda}u_0 + \lambda u_1\|_2 \leqslant \widehat{\lambda}\|u_0\|_2 + \lambda\|u_1\|_2 = 1. \qquad \square$$

Set convexity is preserved under intersection. It is also preserved under *affine transformation*, which is application of a function of the form

$$f : \mathbb{R}^n \to \mathbb{R}^m \ : \ f(x) = Ax + b.$$

In other words, if C is a convex subset of R^n, then the set

$$f(C) := \{f(x) \mid x \in C\},$$

is a convex subset of $\mathbb{R}^m$. What's more, the inverse image of C under f:

$$f^{-1}(C) := \{x \mid f(x) \in C\},$$

is also a convex subset of $\mathbb{R}^n$. Convexity is not however preserved under union, or any other usual operation between sets.

2.2 Definiteness

Let $\mathbb{S}^n$ be the set of square $n \times n$ symmetric matrices:

$$A \in \mathbb{S}^n \Leftrightarrow A^T = A.$$

Definition 8. A symmetric matrix $A \in \mathbb{S}^n$ is called *positive semidefinite* or *nonnegative definite* if for all its eigenvalues λ_i

$$\lambda_i \geqslant 0, \quad i = 0, \ldots, n-1.$$

If the above inequality is strict, then A is called *positive definite*. If $-A$ is positive definite, A is called *negative definite*. If $-A$ is positive semidefinite, then A is called *negative semidefinite*, or *nonpositive definite*.

The set of symmetric positive semidefinite matrices we will denote as $\mathbb{S}^n_+$. The set of positive definite symmetric matrices, we will write as $\mathbb{S}^n_{++}$ (Also, we will note $\mathbb{R}_+$ for the non-negative real numbers and $\mathbb{R}_{++}$ for the positive real numbers).

Lemma 2.

$$A \in \mathbb{S}^n_+ \Leftrightarrow \forall\, x \in \mathbb{R}^n,\ x^T A x \geqslant 0, \tag{2.12}$$

$$A \in \mathbb{S}^n_{++} \Leftrightarrow \forall\, x \in \mathbb{R}^n,\ x^T A x > 0. \tag{2.13}$$

Proof. Let $\lambda_{\min}$ be the minimum eigenvalue of A. We use the property [Boyd and Vandenberghe, 2004, page 647] that for every $x \in \mathbb{R}^n$

$$\lambda_{\min} x^T x \leqslant x^T A x.$$

Then if $\lambda_{\min}$ is positive, so will $x^T A x$ for all $x \in \mathbb{R}^n$. □

Definition 9. The *Gram* matrix associated with $v_0, \ldots, v_{m-1}$, $v_i \in \mathbb{R}^n$, $i = 0, \ldots, m-1$ is the matrix

$$G = V^T V, \quad V = [v_1 \ \cdots \ v_{m-1}],$$

so that $G_{ij} = v_i^T v_j$.

Theorem 1. *Every Gram matrix G is positive semidefinite and for every positive semidefinite matrix A, there is a set of vectors $v_0, \ldots, v_{n-1}$ such that A is the Gram matrix associated with those vectors.*

Proof. ($\Rightarrow$)Let $x \in \mathbb{R}^n$:

$$x^T G x = x^T (V^T V) x = (x^T V^T)(V x) = (V x)^T (V x) \geqslant 0.$$

($\Leftarrow$) For this direction we use the fact that every positive semidefinite matrix A has a positive semidefinite "square root" matrix B such that $A = B^2$ [Horn and Johnson, 1985, pages 405,408]. Then A will be the Gram matrix of the columns of B. □

Solution set of a quadratic inequality

This example gives us good reason to present the following

Theorem 2. *A closed set C is convex if and only if it is* midpoint convex*, which means that it satisfies* (2.1) *for $\lambda = 1/2$.*

Proof. One direction is obvious. If C is convex, then it will be midpoint convex. For the other direction, suppose C is closed and midpoint convex. Then, for every point in C, the midpoint also belongs in C. Now take two points in $a, b \in C$ and let $[a, b]$ be the closed interval from a to b, taken on the line that connects the two points, with positive direction from a to b. We want to prove that $x \in C \ \forall x \in [a, b]$. Take the following sequence of partitions of $[a, b]$

$$P_n = \left\{ \bigcup_{k=0}^{2^{n-1}} \left[\widehat{\frac{k}{2^n}} a + \frac{k}{2^n} b, \widehat{\frac{k+1}{2^n}} a + \frac{k+1}{2^n} b \right] \right\}.$$

All lower and upper bounds of the closed intervals in P_n belong to C, from midpoint convexity. Take any point $x \in [a, b]$. So $x = \widehat{\lambda} a + \lambda b$ for some $\lambda \in [0, 1]$. Then there is a closed interval in P_n, of length $1/2^n$ that contains x. Now take the sequence of points in C,

$$x_n = \{ \frac{k}{2^n} a + \widehat{\frac{k}{2^n}} b \big| x \in [\frac{k}{2^n} a + \widehat{\frac{k}{2^n}} b, \frac{k+1}{2^n} a + \widehat{\frac{k+1}{2^n}} b] \}.$$

If we let $n \to \infty$, then $x_n \to x$. Since C is closed, then it must contain its limit points. Thus $x \in C$. □

We shall use this to prove that the set

$$C = \{x \in \mathbb{R}^n \mid x^T A x + b^T x + c \leqslant 0\},$$

where $b \in \mathbb{R}^n$ and $\in \mathbb{S}^n$, is convex for positive semidefinite A.

Proof. Since C is closed, we will prove that it is midpoint convex and then use Theorem 2, to show that it is convex. Let $f(x) = x^T A x + b^T x + c$. Let $x_0, x_1 \in C \Rightarrow f(x_0), f(x_1) \leqslant 0$. All we need to show is that the midpoint

$(x_0 + x_1)/2 \in C$, or $f((x_0 + x_1)/2)) \leqslant 0$.

$$\begin{aligned} f((x_0 + x_1)/2) &= \frac{1}{4}(x_0 + x_1)^T A(x_0 + x_1) + \frac{1}{2}b^T(x_0 + x_1) + c \\ &= \frac{1}{4}x_0^T Ax_0 + \frac{1}{4}x_1^T Ax_1 + c + \frac{1}{2}x_0^T Ax_1 \\ &= \frac{1}{2}(f(x_0) + f(x_1)) + \frac{1}{2}x_0^T Ax_1 - \frac{1}{4}x_0^T Ax_0 - \frac{1}{4}x_1^T Ax_1 \\ &= \frac{1}{2}(f(x_0) + f(x_1)) - \frac{1}{4}(x_0 - x_1)^T A(x_0 - x_1) \\ &\leqslant 0, \end{aligned}$$

if $A \in \mathbb{S}_+^n$ □

2.3 Cones

Definition 10. A set Ω is called a *cone*, if

$$\forall x \in \Omega,\ \lambda \geqslant 0 \quad \lambda x \in \Omega. \tag{2.14}$$

If in addition, Ω is a convex set, then it is called a *convex cone*.

Lemma 3. *A set Ω is a convex cone if and only if*

$$\forall x_0, x_1 \in \Omega, \lambda_0, \lambda_1 \geqslant 0 \quad \lambda_0 x_0 + \lambda_1 x_1 \in \Omega, \tag{2.15}$$

Proof. ($\Rightarrow$)Define

$$y := \frac{\lambda_0}{\lambda_0 + \lambda_1}x_0 + \frac{\widehat{\lambda_0}}{\lambda_0 + \lambda_1}x_1 = \frac{\lambda_0}{\lambda_0 + \lambda_1}x_0 + \frac{\lambda_1}{\lambda_0 + \lambda_1}x_1.$$

Ω is convex, thus $y \in \Omega$. Also, Ω is a cone, thus

$$(\lambda_0 + \lambda_1)y = \lambda_0 x_0 + \lambda_1 x_1 \in \Omega$$

($\Leftarrow$) If (2.15) is true, then if we choose $\lambda_1 = 0$ we have shown that Ω satisies the cone property. And if we choose any $\lambda_0 \in [0, 1]$ and $\lambda_1 := \widehat{\lambda}_0$, we see that Ω also satisfies the convex property. □

Similarly, we can show that every combination with non-negative coefficients of points of a cone, also lies in the cone. Such combinations are called *conic* or *non-negative linear* combinations. The set of all possible conic combinations of points in a set S, is called the *conic hull* of S. And similarly

with the convex hull of S, it is smallest convex cone that contains S. For example, in $\mathbb{R}^2$, the conic hull of the circle with centre (1,1) and radius 1, is the first quadrant.

A very important example of a convex cone is the *quadratic* or *second-order cone*

$$C = \{(x, t) \in \mathbb{R}^{n+1} \mid \|x\|_2 \leqslant t\}. \tag{2.16}$$

The proof follows a simple application of the triangular inequality. Another important example of a convex cone, is the set $\mathbb{S}^n_+$: the set of symmetric positive semidefinite matrices.

Proof. Let $\lambda_0, \lambda_1 \in \mathbb{R}_+$ and $A_0, A_1 \in \mathbb{S}^n_+$.

$$x^T(\lambda_0 A_0 + \lambda_1 A_1)x = \lambda_0 x^T A_0 x + \lambda_1 x^T A_1 x \geqslant 0 \Rightarrow$$
$$\lambda_0 A_0 + \lambda_1 A_1 \in \mathbb{S}^n_+.$$

□

Definition 11. Let Ω be a convex cone that also satisfies the following properties:

1. Ω is closed.
2. Ω is *solid*, in other words, it has non-empty interior.
3. If $x \in \Omega$ and $-x \in \Omega$ then $x = 0$. (Ω is *pointed*).

Then we say that Ω is a *proper cone*.

Proper cones are very important to the definition of a partial ordering on $\mathbb{R}^n$, or even on $\mathbb{S}^n$. That is a non-trivial problem without a unique optimal solution. There is also another type of cone we will be interested in further on.

Definition 12. Let Ω be a cone. The set

$$\Omega^* = \{y \mid x^T y \geqslant 0 \text{ for all } x \in \Omega\}, \tag{2.17}$$

is called the *dual* cone of Ω.

The dual cone is very useful, as it has many desirable properties [Boyd and Vandenberghe, 2004]:

1. Ω^* is closed and convex.
2. $\Omega_0 \subseteq \Omega_1 \Rightarrow \Omega_0^* \supseteq \Omega_1^*$.
3. If Ω has nonempty interior, then Ω^* is pointed.
4. If the closure of Ω is pointed, then Ω^* has nonempty interior.
5. Ω^* is the closure of the convex hull of Ω.

2.4 Generalised inequalities

As already mentioned, the purpose for introducing the notion of cones, was to be able to define a partial ordering for vectors and matrices. This way, we will be able to give meaning to expressions like *matrix A is less than matrix B*. The problem of defining a partial ordering in such spaces does not have a unique or even best solution. Here, we will give the partial ordering that serves our needs, that is the needs of convex optimization.

Definition 13. Let $\Omega \subseteq \mathbb{R}^n$ be a proper cone. Then the *generalized inequality associated with* Ω will be defined as

$$x \preccurlyeq_\Omega y \Leftrightarrow y - x \in \Omega. \tag{2.18}$$

We will manipulate the above ordering, as we do with ordering on real numbers. We will write

- $x \succcurlyeq_\Omega y \Leftrightarrow x - y \in \Omega$.
- $x \prec_\Omega y \Leftrightarrow y - x \in \mathring{\Omega}$, where $\mathring{\Omega}$ is the interior of the set Ω. In other words, Ω without the boundary.

We shall call the last relation, the *strict* generalized inequality.

If $\Omega := \mathbb{R}_+$, then we get the normal ordering of the real numbers. That means that the generalized inequality we have defined, is indeed a generalization of the usual inequality in real number. That means that the usual ordering in $\mathbb{R}$ can be considered as a special case of $\preccurlyeq_\Omega$. A very significant detail, however, is that the generalized ordering is a partial ordering, which means that not all elements of $\mathbb{R}^n$ will be necessarily comparable. In order to make use of the ordering we have just defined, we have to define Ω to be the appropriate set for each case. To compare vectors, let $\Omega := \mathbb{R}^n_+$(the nonnegative orthant). Then the generalized inequality will be equivalent to componentwise usual inequality of real numbers.

$$x \preccurlyeq_{\mathbb{R}^n_+} y \Leftrightarrow x_i \leqslant y_i, \quad i = 0, \ldots, n-1.$$

For symmetric matrices, let $\Omega := \mathbb{S}^n_+$; the positive semidefinite cone. In this case

$$A \preccurlyeq_{\mathbb{S}^n_+} B \Leftrightarrow B - A \in \mathbb{S}^n_+.$$

In the last two cases, we will drop the subscript, and imply in each case the appropriate proper cone. This has the immediate advantage, that it equips us with a very compact notation for positive semidefinite matrices.

$$A \succcurlyeq 0 \Leftrightarrow x^T A x \geqslant 0 \quad \forall x \in \mathbb{R}^n. \tag{2.19}$$

2.5 Convex functions

Definition 14. Let $D \subseteq \mathbb{R}^n$. A function $f : D \to \mathbb{R}$ is convex if and only if D is convex and for all $x_0, x_1 \in D$ and $\lambda \in [0,1]$

$$f(\widehat{\lambda} x_0 + \lambda x_1) \leqslant \widehat{\lambda} f(x_0) + \lambda f(x_1). \tag{2.20}$$

We say f is *strictly* convex, if and only if, the above inequality, with $x_0 \neq x_1$ and $\lambda \in (0,1)$, is strict. If $-f$ is (strictly) convex, then we say f is (strictly) *concave*. The above inequality is also called **Jensen's inequality**.

Visually, a function is convex if the chord between any two points of its graph is completely above the graph. There are cases of course, that the graph of the function is a line, or a plane, or a multidimensional analogue. In this case, (2.20) holds trivially as an equation. What's more, it holds for $-f$ as well. These functions are called *affine functions* and have the form

$$f : \mathbb{R}^n \to \mathbb{R}^m : f(x) = Ax + b, \tag{2.21}$$

where $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$, and for the reason we just explained, are both convex and concave.

In order to give an immediate relation between convex sets and convex funcitons, we need the following:

Definition 15. The *epigraph* of a function $f : \mathbb{R}^n \supseteq D \to \mathbb{R}$, is the set defined as

$$\mathbf{epi}\, f = \{(x,t) \mid x \in D, f(x) \leqslant t\} \subseteq \mathbb{R}^{n+1}.$$

The set $\mathbb{R}^{n+1} \setminus \mathbf{epi}\, f$, in other words the set

$$\mathbf{hypo}\, f = \{(x,t) \mid x \in D, f(x) > t\} \subseteq \mathbb{R}^{n+1}.$$

is called the *hypograph* of f.

Sometimes, nonstrict inequality can be allowed in the definition of the hypograph of f. What's important, is that the epigraph is the set of all points that lie "above" the graph of f and the hypograph is the set of all points that lie "below" the graph of f. *A function f is convex, if and only if its epigraph is a convex set. And f is concave, if and only if its hypograph is a convex set.*

To determine convexity of functions, it is more efficient to use calculus, than the definition of convexity (under the hypothesis of course, that the function is differentiable). We can use the simple observation, that if a function is convex, then the graph will always be above the tangent line, taken at any point of the domain of the function. To put it rigorously

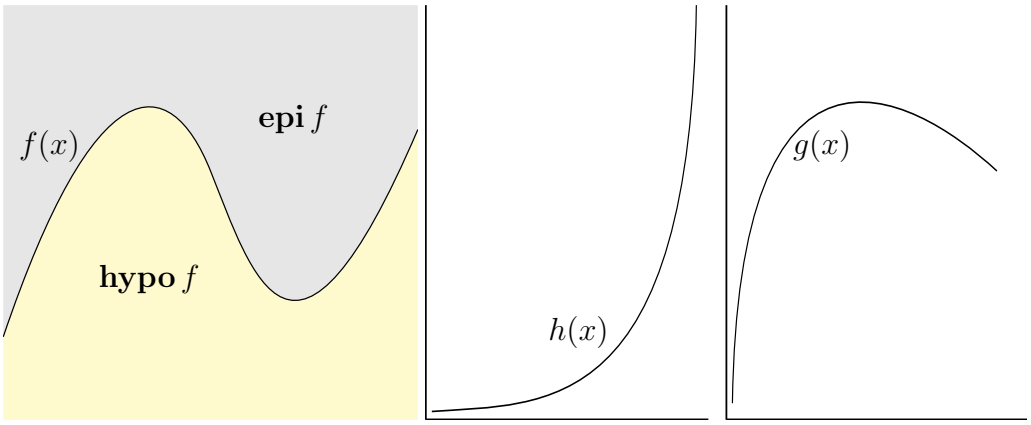

Figure 2.2: Left: Epigraph and hypograph of a function. Right: The graph of a convex (h) and a concave (g) function

Theorem 3. *A differentiable function $f : \mathbb{R}^n \supseteq D \to \mathbb{R}$ is convex, if and only if D is a convex set and*

$$f(y) \geqslant f(x) + \nabla f(x)^T (y - x), \tag{2.22}$$

for all $x, y \in D$.

Equation (2.22) is also called the first order condition. In case that f is also twice differentiable, the second order condition can also be used.

Theorem 4. *Let $\mathcal{H} = \nabla^2 f(x)$, be the* Hessian *of the twice differentiable function $f : \mathbb{R}^n \supseteq D : \mathbb{R}$. Then f is convex if and only if D is a convex set, and*

$$\mathcal{H} \succcurlyeq 0. \tag{2.23}$$

In one dimension, the first and second order conditions, become

$$f(y) \geqslant f(x) + f'(x)(y - x), \tag{2.24}$$
$$f''(x) > 0. \tag{2.25}$$

Operations that preserve convexity

Here we present briefly a list of the most important operations that preserve convexity. Some are obvious, for the rest the proof can be found in Boyd and Vandenberghe [2004].

- *Nonnegative weighted sums:* Let f_i be convex functions and $w_i \geqslant 0$ for $i = 0, \ldots, m-1$. Then g is a convex function where

$$g = w_0 f_0 + \cdots + w_{m-1} f_{m-1}.$$

- *Pointwise maximum and supremum:* . Let f_i be convex functions, $i = 0, \ldots, m-1$. Then g is a convex function, where

$$g = \max\{f_0, \ldots, f_{m-1}\}.$$

- *Scalar composition rules:* (For all the results below, we suppose that $g(x) \in \mathbf{dom}\, f$.) Let $f(x)$ be a convex function. Then $f(g(x))$ is a convex function if
  - $g(x)$ is an *affine* function. That is a function of the form $Ax + b$.
  - g is convex and f is nondecreasing.
  - g is concave and f is nonincreasing.

  The rules for vector composition can be constructed if we apply the above to every argument of f. Also we can find the rules that preserve concavity by just taking the dual (in the mathematical logic sense) propositions of the above (minimum instead of maximum, decreasing instead of increasing, concave instead of convex, etc.).

Chapter 3

Convex optimization

First of all, we introduce the basic concept behind mathematical optimization. An *optimization problem*, or *program* is a mathematical problem of the form

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leqslant b_i, \quad i = 1, \dots m. \end{array} \tag{3.1}$$

We call $f_0(x)$ the *objective function*, $f_i(x)$ the *constraint functions* and x the optimization variable. b_i are normally called the limits or bounds for the constraints. Normally, $f_i : D_i \to \mathbb{R}$, where $D_i \subseteq \mathbb{R}^n$, for $i = 0, \dots m \in \mathbb{N}$. If the domains D_i are not the same set for all i, then we look for a solution in the set $D = \cup_{i=0}^{n-1} D_i$, under the condition of course that this is a nonempty set. If D is nonempty, any $x \in D$ that satisfies the constraints is called a *feasible point*. If there is at least one such x, then (3.1) is called *feasible*. Otherwise, it is called *infeasible*. In case $D_0 = \mathbb{R}^n$ and $\forall x \in D_0, f_0(x) = c$, $c \in \mathbb{R}$, then any x minimizes f_0 trivially, as long as (3.1) is feasible. In that case, we only have to check that the constraints are consistent. In other words, verify that D is non-empty and then solve the following:

$$\begin{array}{ll} \text{find} & x \\ \text{subject to} & f_i(x) \leqslant b_i \quad i = 1, \dots, m. \end{array} \tag{3.2}$$

This is called a *feasibility problem*. And any solution to (3.2), will be a feasible point and vice versa. If we denote S, the solution set to (3.2), then the set $F = S \cap D_0$ is called the *feasible set* of (3.1).

In the case that the constraint functions are omitted, then the problem is called an *unconstrained* optimization problem. Also, for some $i > 0$, there is the possibility that instead of the inequality, we have the equality $f_i(x) = b_i$. If we incorporate the constraint bounds into the constraint

function and also rename appropriately, we can rewrite (3.1) in what is called *standard form*:

$$\begin{array}{lll} \text{minimize} & f(x) & \\ \text{subject to} & g_i(x) \leqslant 0 & i = 0, \ldots, p-1 \\ & h_i(x) = 0 & i = 0, \ldots m-1. \end{array} \tag{3.3}$$

Now that we have the problem in standard form, we can express its solution, or *optimal value* f^* as

$$f^* = \inf\{f(x) \mid g_i(x) \leqslant 0, i = 0, \ldots, p-1,\ h_i(x) = 0, i = 0, \ldots, m-1\}.$$

We call the x^*, such that $f(x^*) = f^*$ the *optimal point*, or the *global optimum*. Of course, in this general case, there is a possibility that we also have *local optimal points*. Which means that there is $\varepsilon > 0$ such that

$$f(x) = \inf\{f(z) \mid g_i(z) \leqslant 0, i = 0, \ldots, p-1, h_i(z) = 0, i = 0, \ldots, m-1, \\ \|z - x\| \leqslant \varepsilon.\} \tag{3.4}$$

Definition 16. Consider the optimization problem in standard form (3.3). Suppose that the following conditions are satisfied:

- The objective function is convex; in other words, f satisfies Jensen's inequality (2.20).
- The inequality constraints $g_i, i = 0 \ldots m-1$ are also convex functions.
- The equality constraints are *affine* functions; that is functions of the form $h_i(x) = A_i x + b_i, i = 0 \ldots p-1, A \in \mathbb{R}^{s \times n}, b \in \mathbb{R}^s$.

Then (3.3) is called a *convex optimization problem* (in standard form) or COP.

Theorem 5. *Consider any convex optimization problem and suppose that this problem has a local optimal point. Then that point is also the global optimum.*

Proof. [Boyd and Vandenberghe, 2004] Let x be a feasible and local optimal point for the COP (3.3). Then it satisfies (3.4). If it is not global optimum, then there is feasible $y \neq x$, and $f(y) < f(x)$. That means that $\|y - x\| > \varepsilon$, otherwise, we would have $f(x) \leqslant f(y)$. Consider the point

$$z = \widehat{\lambda} x + \lambda y, \qquad \lambda = \frac{\varepsilon}{2\|y - x\|}.$$

Since the objectives and the constraints are convex functions, they have convex domains. By Lemma 1, the feasible set F of (3.3) is a convex set. Thus, z is also feasible. And by definition,

$$\|z - x\| = \varepsilon/2 < \varepsilon,$$

so $f(x) \leqslant f(z)$. But Jensen's inequality for f gives

$$f(z) \leqslant \widehat{\lambda} f(x) + \lambda f(y) < f(x)$$

(because $f(y) < f(x)$). By contradiction, x is the global optimum. □

This fact makes algorithms for solving convex problems much more simple, effective and fast. The reason is that we have very simple optimality conditions, especially in the differentiable case.

Theorem 6. *Consider the* COP (3.3)*. Then x is optimal, if and only if x is feasible and*

$$\nabla f(x)^T (y - x) \geqslant 0, \tag{3.5}$$

for all feasible y.

Proof. ($\Rightarrow$) Suppose x satisfies (3.5). Then if y is feasible, by (2.22) we have that $f(y) \geqslant f(x)$.
($\Leftarrow$) Suppose that x is optimal, but there is feasible y such that

$$\nabla f(x)^T (y - x) < 0.$$

Then consider the point $z = \widehat{\lambda} + \lambda y$, $\lambda \in [0, 1]$. Since the feasible set is convex, then z is feasible. Furthermore

$$\frac{d}{d\lambda} f(z)\bigg|_{\lambda=0} = \nabla f(x)^T (y - x) < 0,$$

which means that for small enough, positive λ, $f(z) < f(x)$. That is a contradiction to the assumption that x was optimal. □

For all the reasons mentioned above, it is only natural to try to extend the use of convex optimization techniques. To succeed that, we use mathematical 'tricks' to transform some non-convex problems, into convex, whenever that is possible.

3.1 Quasiconvex optimization

Definition 17. Let $f : \mathbb{R}^n \supseteq D \to \mathbb{R}$ such that D is convex and for all $\alpha \in \mathbb{R}$, $S_\alpha := \{x \in D \mid f(x) \leqslant \alpha\}$ are also convex sets. Then f is called *quasiconvex*. The sets S_α are called the $\alpha-$sublevel, or just sublevel sets of f. If $-f$ is quasiconvex, then f is called *quasiconcave*. If f is both quasiconvex and quasiconcave, then it is called quasilinear.

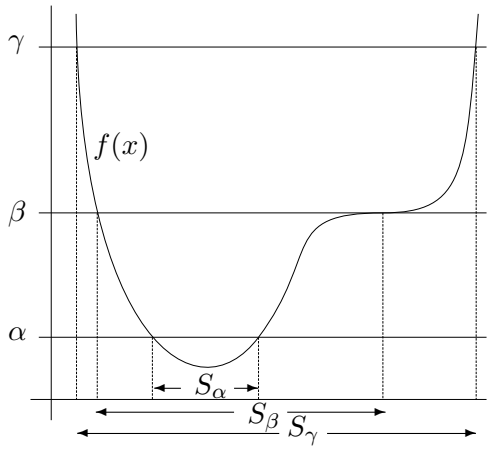

Figure 3.1: A quasiconvex function f.

Quasiconvex functions could be loosely described as "almost" convex functions. However they are not convex, which means that it is possible for a quasiconvex function to have non-global local optima. As a result, we can't solve (3.3), for f quasiconvex, using convex problem algorithms. However, there is a method to approach these problems making use of the advantages of convex optimization, and that is through convex feasibility problems.

Let $\phi_t(x) : \mathbb{R}^n \to \mathbb{R}$, $t \in \mathbb{R}$ such that $\phi_t(x) \leqslant 0 \Leftrightarrow f(x) \leqslant t$ and also ϕ_t is a nonincreasing function of t. We can easily find such an ϕ_t. Here is an outline of how one might construct one:

$$\phi_t(x) = \begin{cases} \text{linear decreasing non-negative}, & x \leqslant \inf S_t, \\ 0, & x \in S_t, \\ \text{linear increasing non-negative}, & x \geqslant \sup S_t. \end{cases}$$

If we also adjust the linear components of ϕ_t, such that it is continuous and nonincreasing in t, then it is easy to see that this is a convex function such as the one we are looking for. Now consider the following feasibility problem

$$\begin{array}{lll} \text{find} & x & \\ \text{subject to} & \phi_t(x) \leqslant 0 & \\ & g_i(x) \leqslant 0 & i = 0, \ldots, p-1 \\ & h_i(x) = 0 & i = 0 \ldots m-1. \end{array} \tag{3.6}$$

We still denote f^* as the optimal value of (3.3). But now we suppose that f is quasiconvex and the restraints are as in the COP. Now, if (3.6) is feasible, then $f^* \leqslant t$. As a matter of fact $f^* \leqslant f(x) \leqslant t$. If it is not feasible,

```
def quasib(dn,up,eps,feas):
  'Bisection method for quasiconvex optimization'
  while up−dn>eps: # eps is the tolerance level
    t=(dn+up)/2.0
    b=feas(t) # function feas solves the feasibility problem at the midpoint
    if b=='optimal':
      up=t
    else:
      dn=t
  return t
```

Program 3.1: *quasibis.py*

then $f^* \geqslant t$. We can then construct a bisection method, such that starting with a large interval that contains f^*, we can find it (within given tolerance boundaries) by solving a convex feasibility problem and bisecting the initial interval.

Program 3.1 realizes that. After each step in the loop, the interval is bisected. Thus after k steps, it has length $2^{-k}(\text{up} - \text{dn})$. The operation ends when that number gets smaller than ε. So the algorithm terminates when $k > \log_2((\text{up} - \text{dn})/\varepsilon)$.

3.1.1 Golden section search

Bisection is what we need for this operation we describe above. But we can't discuss optimization and not mention the *golden section search*. It is a very efficient technique for finding the extremum of a quasiconvex or quasiconcave function in one dimension. In contrast to the bisection method, at each step the golden section search algorithm involves the evaluation of the function at three steps, and then an additional fourth one. After the control, three points of these four are selected, and the procedure continues.

The algorithm starts with input a closed interval [dn,up], which we know contains the minimum and an internal point g_0. Then find which one of the intervals $[\text{dn}, g_0]$,$[g_0, \text{up}]$ is the largest and evaluate the function at an internal point g_1 of that interval. Then compare the value of the function in the two points, g_0 and g_1 and repeat the step, with one of the triplets (dn, g_1, g_0) or (g_0, up, g_1). To ensure the fastest possible mean convergence time, we choose g_0 as the golden mean of up and dn ($g_0 = \widehat{\theta}\text{dn}+\theta\text{up}$, where $\theta = 2/(3 + \sqrt{5})$). Hence the name of the method. For the same reasons g_1

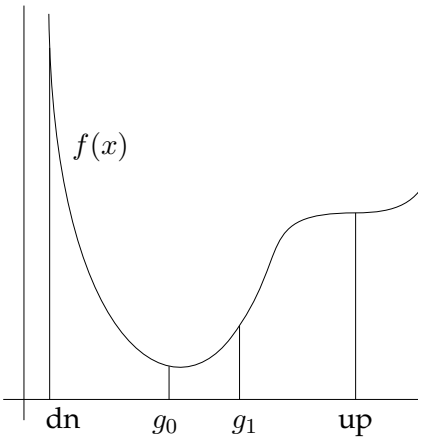

Figure 3.2: Golden section search

will be chosen to be dn $- g_0 +$ up, thus ensuring that up $- g_0 = g_1 -$ dn.

To be more precise take an example where we seek the minimum of a quasiconvex function f (Figure 3.2). We begin with the interval [dn,up], which we know contains the minimum and the internal point g_0. From the choice of g_0, $[g_0, \text{up}]$ will be the largest subinterval. Next we find g_1 and compare $f(g_0)$ to $f(g_1)$. If the former is greater than the latter, repeat the step with $[g_0, \text{up}]$ and g_1. Otherwise repeat with $[\text{dn}, g_1]$ and g_0. The algorithm is an extremely efficient one-dimensional localization method and is closely related to *Fibonacci search*.

3.2 Duality

It is necessary, before introducing the notion of the *dual problem*, to give some basic definitions.

Definition 18. Consider the optimization problem in standard form (3.3). Define the *Lagrange dual function* $g : \mathbb{R}^p \times \mathbb{R}^m \to \mathbb{R}$ as

$$g(\lambda, \nu) = \inf_{x \in D} \left(f(x) + \sum_{i=0}^{p-1} \lambda_i g_i(x) + \sum_{i=0}^{m-1} \nu_i h_i(x) \right), \tag{3.7}$$

where $D = \mathbf{dom}(f) \cup_{i=0}^{p-1} \mathbf{dom}(g_i) \cup_{i=0}^{m-1} \mathbf{dom}(h_i)$. The quantity whose infimum is taken, $L(x, \lambda, \nu) := \big(f(x) + \sum_{i=0}^{p-1} \lambda_i g_i(x) + \sum_{i=0}^{m-1} \nu_i h_i(x)\big)$, is called the *Lagrangian* of the problem. The variables λ and ν are called the *dual variables* and λ_i, ν_i the *Lagrange multipliers* associated with the ith inequality and equality constraint accordingly.

The reason to present the dual function is that it can be used to provide a lower bound for the optimal value f^* of (3.3), for all $\lambda \succcurlyeq 0$ and for all ν.

Proof. Consider a feasible point $\bar{x}$ for (3.3). Then $g_i(\bar{x}) \leqslant 0$ for all $i = 0 \dots p-1$ and $h_i(\bar{x}) = 0$ for all $i = 0 \dots m-1$. Take $\lambda \succcurlyeq 0$. Then

$$\sum_{i=0}^{p-1} \lambda_i g_i(\bar{x}) + \sum_{i=0}^{m-1} \nu_i h_i(\bar{x}) \leqslant 0 \Rightarrow L(\bar{x}, \lambda, \nu) \leqslant f(\bar{x}) \Rightarrow$$

$$g(\lambda, \nu) = \inf L(x, \lambda, \nu) \leqslant L(\bar{x}, \lambda, \nu) \leqslant f(\bar{x}). \qquad \square$$

What's more, since g is the pointwise infimum of a family of affine functions of (λ, ν), it is always concave, even though (3.3) might not be convex. This makes the search for a best lower bound a convex objective.

Definition 19. Consider (3.3), the optimization problem in standard form. Then the problem

$$\begin{array}{ll} \text{maximize} & g(\lambda, \nu) \\ \text{subject to} & \lambda \succcurlyeq 0, \end{array} \tag{3.8}$$

is called the *Lagrange dual problem* associated with (3.3). In that context, the latter will also be referred to as the *primal problem*.

We can see, that for the reasons already mentioned, the dual problem is always convex, even if the primal is not. Thus we can always get a lower bound for the optimal value of any problem of the form (3.3), by solving a convex optimisation problem.

Lemma 4 (Weak Duality)**.** *Let f^* be the optimal solution to the primal problem* (3.3)*, and d^* the optimal solution to the dual problem* (3.8)*. Then, d^* is always a lower bound for f^* : $f* \geqslant d*$.*

We have already proven this property for all values of g. Thus it trivially holds for the maximum value of g. Even more useful is the property that holds when the primal is a convex problem. Then, under mild conditions, we have guaranteed equality and the dual problem is equivalent to the primal. A simple sufficient such condition is *Slater's condition*. That is the existence of a feasible point x, such that $g_i(x) < 0$ for all i such that g_i is not affine. The proof of the next theorem is given in Boyd and Vandenberghe [2004].

Theorem 7 (Strong Duality)**.** *Let* (3.3) *be a convex optimization problem. Let* (3.8) *be its dual. If* (3.3) *also satisfies Slater's condition, then $f^* = d^*$, where f^* and d^* are the optimal values of the primal and the dual problem accordingly.*

Many solvers in optimization software, rely on iterative techniques based on Theorem 7. They compute the value of the dual and the primal objective on an initial point, find the difference between the two and if it is greater than some tolerance level, they continue and evaluate on the next point, which is chosen based on the value of the gradient and the Hessian. The operation terminates when the tolerance level is reached.

3.3 Linear programming

An important class of optimization problems is *linear programming*. In this class of problems, the objective function and the constraints are affine. Thus we have the general form:

$$\begin{array}{ll} \text{minimize} & c^T x + d \\ \text{subject to} & Gx \preccurlyeq h \\ & Ax = b, \end{array} \tag{3.9}$$

where $G \in \mathbb{R}^{p \times n}$, $A \in \mathbb{R}^{m \times n}$, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^p$, $h \in \mathbb{R}^m$ and $d \in \mathbb{R}$. The above is the general form of a *linear program*, and will be referred to as LP. Since affine functions are convex, linear programming can be considered as a special case of convex optimization. However, it has been developed separately, and nowadays we have very efficient ways and algorithms to solve the general LP. Note that the feasible set of an LP is a polyhedron.

Although (3.9) is in standard form as a COP, it is not in LP standard form. The *standard form* LP would be

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax = b \\ & x \succcurlyeq 0. \end{array} \tag{LP}$$

There is also the *inequality form* LP which very often occurs.

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \preccurlyeq b. \end{array} \tag{3.10}$$

Now, given the standard form LP (LP), we can find its dual:

$$\begin{array}{ll} \text{minimize} & b^T y \\ \text{subject to} & A^T y + z = c \\ & z \succcurlyeq 0. \end{array} \tag{3.11}$$

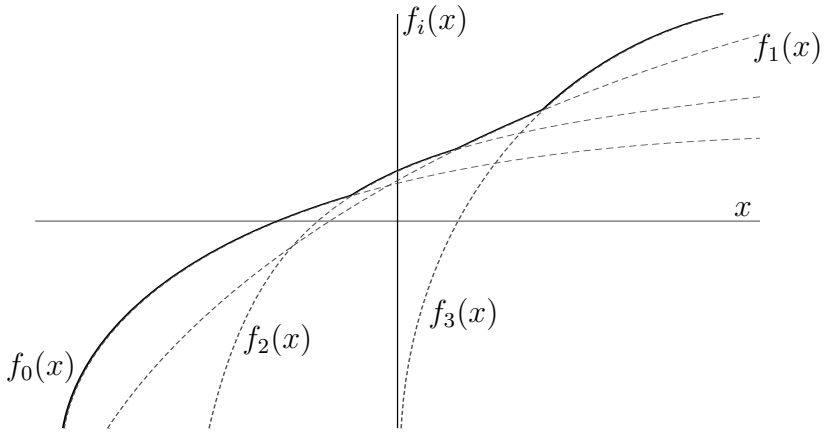

Figure 3.3: The objective function $f_i(x) = \frac{c_i x + d_i}{e_i x + f_i}$ for a one dimensional generalised linear fractional problem. For every value of x, the maximum is taken over all dashed curves. The result is a quasiconvex curve shown in bold.

3.3.1 Linear-fractional programming

If we replace the linear objective in (3.9) with a ratio of linear functions, then the problem is called a *linear-fractional program*.

$$\begin{array}{ll} \text{minimize} & \dfrac{c^T x + d}{e^T x + f} \\ \text{subject to} & e^T x + f > 0 \\ & Gx \preccurlyeq h \\ & Ax = b. \end{array} \tag{LFP}$$

This problem is not convex any more, but quasiconvex. In this case however, there is a more effective approach than the quasiconvex optimization approach. This problem can be transformed to an equivalent LP, with the cost of adding one more extra optimization variable and under the condition that the feasible set of (LFP) is nonempty. Consider the LP

$$\begin{array}{ll} \text{minimize} & c^T y + dz \\ \text{subject to} & Gy - hz \preccurlyeq 0 \\ & Ay - bz = 0 \\ & e^T y + fz = 1 \\ & z \succcurlyeq 0. \end{array} \tag{3.12}$$

This problem is equivalent to (LFP) [Boyd and Vandenberghe, 2004]. Also, if we let x^* and y^*, z^* be the optimal points of these two problems, then

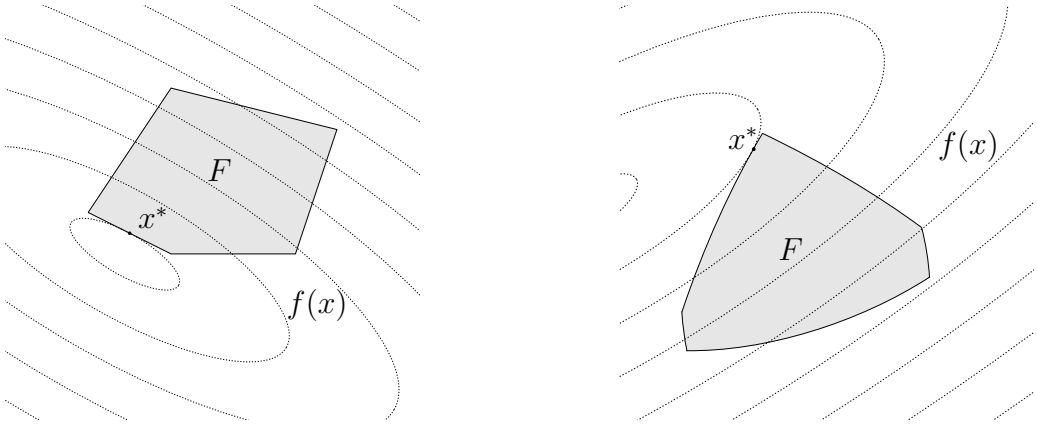

Figure 3.4: Left: Quadratic Program. Right: Quadratically Constraint Quadratic Program. The shaded region is the feasible set F, the thin curves are the contour lines of the objective function f and x^* is the optimal point.

$x^* = y^* + z^*$ and $y^* = x^*/(e^T x^* + f)$ and $z^* = 1/(e^T x^* + f)$.

If we make one small alteration however, this problem becomes much more difficult and no longer equivalent to LP. We can change the objective, into

$$\max_{0 \leqslant i < \rho} \frac{c_i^T x + d_i}{e_i^T x + f_i},$$

which means that we have ρ linear fractional functions, whose maximum we want to minimize. This problem is the *generalised linear-fractional program* and is also quasiconvex. We couldn't however apply the transformation to LP here, as it is possible that for different values of x, we get the maximum for different i (Figure 3.3). Thus, this problem is significantly harder.

3.4 Quadratic programming

If the objective in (3.3) is a convex quadratic function and also the equality constraint functions are affine, then we have a *quadratic program* (QP). If also the equality constraint functions g_i are convex quadratic, then we have a *quadratically constrained quadratic program* (QCQP). A QP and a QCQP

can have the forms

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^T P x + q^T x + r \\ \text{subject to} & Gx \preccurlyeq h \\ & Ax = b, \end{array} \tag{QP}$$

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^T P x + q^T x + r \\ \text{subject to} & \frac{1}{2}x^T P_i x + q_i^T x + r_i, \quad i = 0, \ldots, p-1 \\ & Ax = b, \end{array} \tag{QCQP}$$

accordingly, where $x \in \mathbb{R}^n$, $P, P_0, \ldots, P_{p-1} \in \mathbb{S}^n_+$, $G \in \mathbb{R}^{p \times n}$, $A \in \mathbb{R}^{m \times n}$, $h \in \mathbb{R}^p$, $b \in \mathbb{R}^m$, $q \in \mathbb{R}^n$, $r \in \mathbb{R}$.

A very well-known example of QP is the *least-squares problem* or *regression analysis*, where the objective function is $\|Ax - b\|_2^2$. However, this unconstrained minimization problem does have an analytic solution. That is $x = Cb$, where

$$C = V \Sigma^{-1} U^T$$

and $U \Sigma V^T$ is the *singular value decomposition* of A [Boyd and Vandenberghe, 2004, A.5.4].

3.4.1 Second-order cone programming

A generalization of the QCQP is the problem

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & \|A_i x + b_i\|_2 \leqslant e_i^T + d_i, \quad i = 0, \ldots, p-1 \\ & Fx = g, \end{array} \tag{SOCP}$$

where $x \in \mathbb{R}^n$, $A_i \in \mathbb{R}^{n_i \times n}$, $b_i \in \mathbb{R}^{n_i}$, $c, e_i \in \mathbb{R}^n$, $d_i \in \mathbb{R}$, $F \in \mathbb{R}^{m \times n}$ and $g \in \mathbb{R}^m$. Problem (SOCP) is called a *second-order cone program*. The name is due to the fact that we require $(A_i x + b_i, e_i^T x + d_i)$ to lie in the second order cone (2.16) for $i = 0, \ldots, p-1$.

3.5 Semidefinite programming

We can also allow generalised matrix inequalities for our constraints. An optimization problem with linear objective and affine constraints including equalities, inequalities and generalised inequalities, is called a *conic*

form problem or a *cone program*. We can easily include all inequality constraints into one generalised matrix inequality constraint using simple linear algebra. So we can write the general cone program as

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Fx + g \preccurlyeq_\Omega 0 \\ & Ax = b, \end{array} \tag{3.13}$$

where c, g, b are the vector parameters of the problem and F, A are matrices and Ω is a proper cone. Since proper cones are convex sets, this problem remains a convex optimization problem.

In the case that $\Omega = \mathbb{S}^n_+$, the positive semidefinite cone, then (3.13) is called a *semidefinite program* or just SDP. In semidefinite programming, the inequality constraint can be interpreted as requiring for a symmetric matrix to be positive semidefinite. The standard form for the SDP is

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & x_0 F_0 + x_1 F_1 + \cdots + x_{n-1} F_{n-1} + G \preccurlyeq 0 \\ & Ax = b, \end{array} \tag{SDP}$$

where $F_0, F_1, F_{n-1}, G \in \mathbb{S}^n$, $A \in \mathbb{R}^{m \times n}$ and $b, c \in \mathbb{R}^n$. We could see the SDP as a generalisation of the LP. As a matter of fact, if $F_0, \ldots, F_{n-1}, G$ are all diagonal, then (SDP) is a LP with n linear inequality and m linear equality constraints.

There has been much software developed for solving SDP. Due to the nature of the problem, these solvers very efficient and reliable [Benson et al., 2000, Benson and Ye, 2008, Vandenberghe et al., 1998].

There are other types of convex optimization problems, like *geometric programming*, which we do not mention as we are not considering any applications of these types in this report.

Chapter 4

Applications

In this chapter, we are presenting a set of real optimization problems that we try to understand better and if possible solve with our python tools. We are attempting a new perspective on these problems, which will hopefully lead to better ways of solving them.

4.1 The Virtual Data Center problem

This problem was suggested to us by members of the BT ICT Research team as a yet unsolved problem related to a service which is still in the planning process: *Infrastracture as a Service.* We consider M servers, providing services via a network to N users. These servers could be *Data Centers* equipped with powerful hardware and the users can be customers, who can use part of that equipment for their own business, for a lower cost than buying the equipment on their own. Information is being transferred through a network. There is a cost for every user to connect to any server, which could be latency, or any other performance measure. The objective is to assign users to servers, in a way that maximizes performance by minimizing the connection cost. There are capacity constraints on the servers. For simplicity, we assume that all users take up the same amount of resources, so the constraint will be on the maximum amount of users a server can be connected to. Another constraint, is that only one server can be connected to each user. We don't allow for a user to be connected to more than one server.

In case that $M = N$, this is the **Assignment problem**, where N people have to be assigned to do N jobs with cost α_{ij} for person i to do job j. If we were to allow any number of users to be connected to any number of servers, this would be the general case of the **Transportation Problem**

[Garvin, 1960].

But let's formulate this problem rigorously. Let C be the cost matrix we described above. C will be $M \times N$ and c_{ij} will be the cost for user j to be connected to server i. Our optimization variable X, will be also an $M \times N$ matrix with elements 0, or 1. If element ij is 1, that will imply a connection between server i and user j. There is also an M-dimensional vector a, whose i^{th} element is the capacity of server i. Our objective then will be to minimize the sum over all rows of $C^T X$, which is the total connection cost, under the condition that server i is connected to a_i users at most.

In order to be consistent with the LP format and also to implement the problem more easily, instead of the matrices C and X, we will use c, x which are both $MN \times 1$ and are the same as the original problem parameters, only "flattened". To be more precise, the ij^{th} element of C will be the $(i \bmod M + j(N-1))^{\text{th}}$ element of c, and the same for X and x.

$$C = \begin{bmatrix} c_{00} & c_{01} & \cdots \\ c_{10} & \ddots & \\ \vdots & & c_{MN} \end{bmatrix}, X = \begin{bmatrix} x_{00} & x_{01} & \cdots \\ x_{10} & \ddots & \\ \vdots & & x_{MN} \end{bmatrix} \longrightarrow c = \begin{bmatrix} c_{00} \\ c_{10} \\ \vdots \\ c_{M0} \\ c_{01} \\ \vdots \\ c_{M1} \\ \vdots \end{bmatrix}, x = \begin{bmatrix} x_{00} \\ x_{10} \\ \vdots \\ x_{M0} \\ x_{01} \\ \vdots \\ x_{M1} \\ \vdots \end{bmatrix}.$$

This formulation is common in literature and the notation used is

$$c = \mathbf{vec}(C), x = \mathbf{vec}(X). \tag{4.1}$$

We will use this notation to describe the above transformation from here forth. After that formulation, the objective will just be $c^T x$ and we can now give the problem in standard LP form. To keep things as simple as possible we will denote $k(i,j) = i \bmod M + j(N-1)$:

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & \sum_{0 \leqslant i < M} x_{k(i,j)} = 1, \quad j = 0, \ldots, N-1 \\ & \sum_{0 \leqslant j < N} x_{k(i,j)} \leqslant a_i, \quad i = 0, \ldots, M-1 \\ & x \geqslant 0 \end{aligned} \tag{4.2}$$

The first constraint simply says that exactly one element of every column of X must be non-zero, which means that exactly one connection is allowed per user. The second is just the capacity constraint on the data

```
from cvxopt.base import matrix,spmatrix
from cvxopt.solvers import lp

def VDCopt(n_DC,n_users,cost,capacity):
  ln,p,dict=len(capacity),n_DC*n_users,{}
  rindx=[0]*p+[[i+1]*n_DC for i in xrange(n_users-1)]
  rindx[p:]=[rindx[p:][i][j] for i in xrange(n_users-1) for j in xrange(n_DC)]
  cindx=range(p)+[range(n_DC*(i+1))[i*n_DC:(i+1)*n_DC] for i in xrange(n_users)]
  cindx[p:]=[cindx[p:][i][j] for i in xrange(n_users-1) for j in xrange(n_DC)]
  A=spmatrix(1.0,rindx,cindx)
  b=matrix(1.0,(n_users,1))
  b[0]=n_users
  capr=[[p+i]*n_users for i in xrange(ln)]
  caprindx=[capr[i][j] for i in xrange(ln) for j in xrange(n_users)]
  capc=[range(p)[i::n_DC] for i in range(ln)]
  capcindx=[capc[i][j] for i in xrange(ln) for j in xrange(n_users)]
  G=spmatrix([-1.0]*p+[1.0]*(ln*n_users),range(p)+caprindx,range(p)+capcindx)
  h=matrix([0.0]*p+capacity,(p+ln,1))
  a=lp(cost,G,h,A,b,solver='glpk')['x']
  dict['Optimal value']=(cost.T*a)[0]
  for i in xrange(p):
    if a[i]>0.9:        # give solution in the form "u_j:(DC_i,cost_ij)"
      dict['u_%d'%(i/n_DC,)]=('DC_%d'%(i%n_DC,),cost[i])
  return dict

if __name__=='__main__':
  cost=matrix([10.0, 22.0, 12.0, 21.0, 15.0, 25.0]) # [c00,c10,c01,c11,c02,c12]
  capacity=[2,2]
  print VDCopt(2,3,cost,capacity)
```

Program 4.1: *VDCopt.py*: A simplified function *VDCopt* which solves the VDC problem. Its arguments are the number of users and data centers, a matrix "cost" which is as $\mathbf{vec}(C)$ in (4.1) and a list of integers, for the capacity of each data center. In this example we have 3 users and 2 data centers with capacity 2 for each. Although it would be cheaper for all the users to be connected to the first data center (with corresponding costs 10,12 and 15), the capacity constraint forces the solution: {**'Optimal value': 46.0, 'u_2': ('DC_0', 15.0), 'u_1': ('DC_1', 21.0), 'u_0': ('DC_0', 10.0)**}

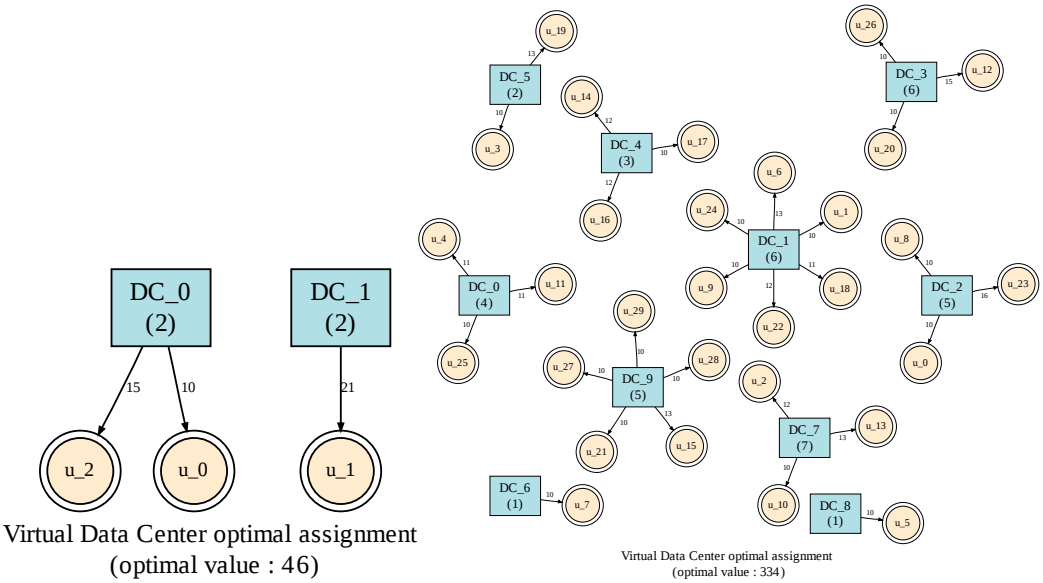

Figure 4.1: The solution for the virtual data center problem. Left: the small example of Program 4.1. Right: a bigger example with 10 data centers and 30 users. The numbers in brackets denote the capacity of every data center. The labels on the edges denote the cost of connection between data center i and customer j.

centers. We don't need to add any additional constraints to make x a 0,1 vector, as the integer right hand sides of the constraints will force an integer optimal solution (proof in [Garvin, 1960] pages 92,116) given that we use the simplex method to calculate the solution. That is ensured by using the *'glpk'* optional solver in **CVXOPT** [Makhorin, 2003]. Then, the positive constraint, and the constraint of all elements in one column adding up to 1, will allow any element to be either 0 or 1.

Program 4.1 gives a simple function *VDCopt* which solves (4.2) for a small example and returns a dictionary with the assignments and the value of the minimal cost. In our examples, we are using connection cost values to be integers in the range of 10 and 25, which is realistic enough if the cost is latency measured in *hops* (*the number of intermediate routers, or points between the source and the target of transmission*) and the internet, or any other extended network is used to provide the connections.

4.1.1 Scalability

An important question to ask about such a network is: how does it scale, as the number of users grow? In other words, we would like to know for how big a number of users, can our optimum configuration be considered satisfactory? That depends first of all on the characteristics of the network itself.

We have considered two cases. First, we consider the case where the capacities of the data centers came from a uniform-like distribution. In the second, we took the extreme case of having one big data center with large capacity and all the others were very small in comparison and practically equivalent to each other. In both cases we range the number of users from the same number as the data centers to the maximum capacity of the network. The results are shown in Figure 4.2.

What we can conclude from the results, is that in the first case, the network corresponds extremely well to the increasing load of users. In plain terms, that means that the provider can guarantee performance to the customer. No matter how many users are added, even when the capacity of the network is reached, the user will never understand the difference.

In the second case, we see a completely different picture. For a small number of users, the picture is similar to the previous case. However, there seems to be a critical number and when exceeded, the average cost, or cost per user increases dramatically. How soon that critical number will be reached, depends of course on the number of data centers. In the bottom right example in Figure 4.2, it may not even be reached even when the number of users exceeds 3000, although eventually it will.

What the above describe could be two different situations. We can imagine a company providing the data center service, trying to decide how to set up their network. One choice would be to decide beforehand how big the capacity of the network will ever have to be and distribute that almost evenly among their data centers. The other choice, would be to build one data centers in a place with high demand, at that present time, like an industrial city and small ones spread out on a wider area. The large data center could be possibly upgraded in the future to cope with an increase in the number of customers. At that time, that might be the cheapest solution, but as the number of customers grew, the first set-up would be the most reliable and efficient.

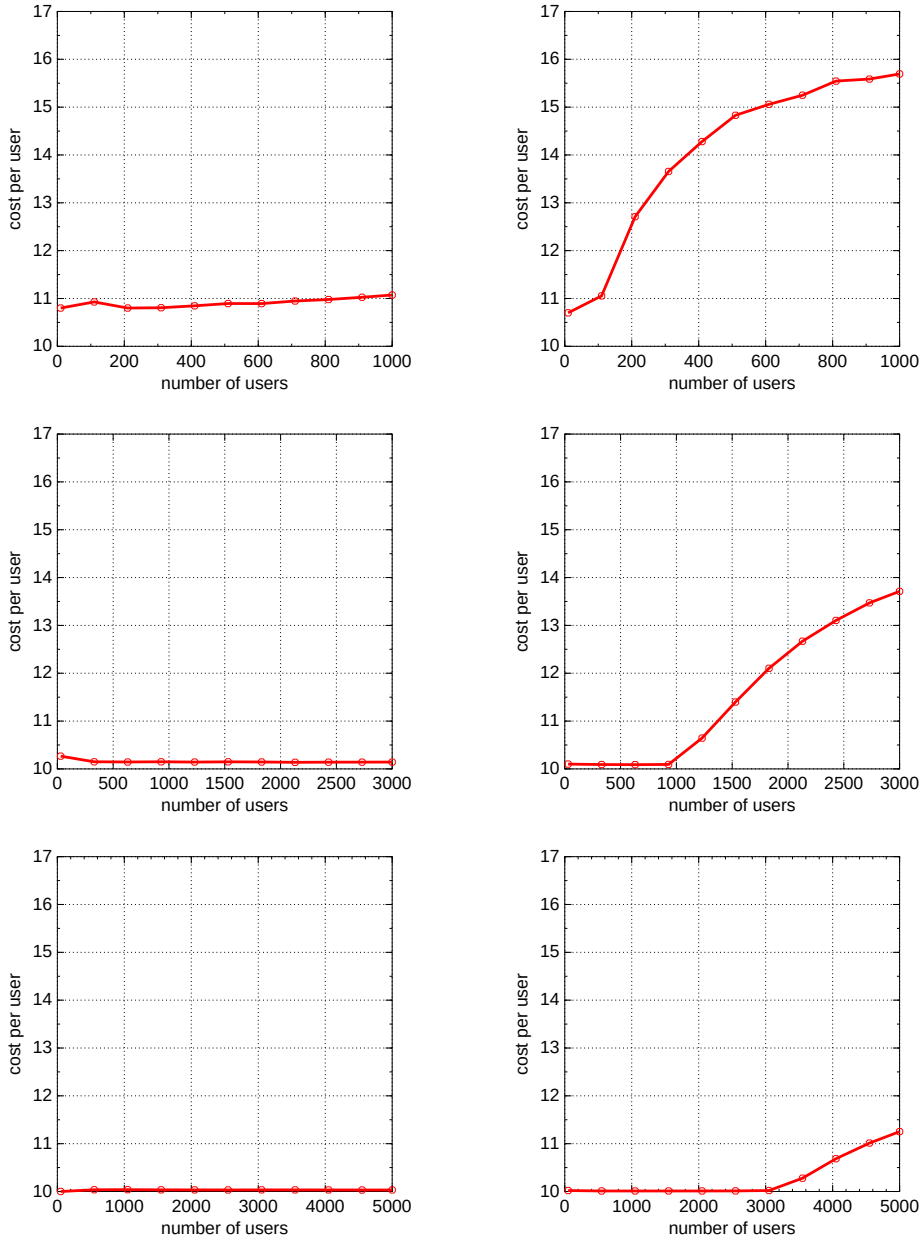

Figure 4.2: Scalability of the virtual data center problem. Top: 10 data centers and users ranging from 10 to 1000. Middle: 30 data centers and users ranging from 30 to 3000. Bottom: 50 data centers and users ranging from 50 to 5000. Left: Size of data centers uniformly distributed. Right : Most of the total network capacity is concentrated on one data center and the rest is evenly distributed among the rest.

4.2 The Lovász ϑ function

We are interested in calculating the clique and the chromatic number of a given graph. Finding the chromatic number of a graph is the same problem as finding the minimum number of colours to be used in a map, so that no two countries that share borders will have the same colour. Hence the name "chromatic". Another problem related with the chromatic number, would be the least number of channels to be used in a wireless communications network, so that there will be no interference in the network.

4.2.1 Graph theory basics

First of all, let's introduce some basic concepts about graphs. Intuitively, a graph is a set of nodes, some pairs of which are connected. Images come to mind, varying from two connected dots, to complex representations of networks (see Figure 4.3).

Definition 20. Let V be a set. Let $E \subseteq \mathcal{V}$, where $\mathcal{V}$ is the set of all possible pairs of elements of V. The ordered pair (V, E), is called the *graph* (or *undirected graph*) with vertices (or nodes) V and edges E. If the elements of $\mathcal{V}$ are ordered pairs, then (V, E) is called a *directed graph*. We denote $\mathcal{G}$ for the set of all graphs. (*Note*: Here, we will only discuss undirected graphs and for a finite set of vertices. So we will drop the characterization "undirected" and imply it whenever we write "graph".)

Definition 21. An *induced subgraph* of (V, E) is an ordered pair (U, D), where $U \subseteq V$, $D \subseteq E \cap \mathcal{U}$ and $\mathcal{U}$ is the set of all possible pairs of elements of U. (*Note*: Again, we are only going to consider induced subgraphs, so we will drop the characterisation "induced", and just write "subgraph". We will also write $F \subseteq G$ if and only if F is an induced subgraph of G.)

We have

$$\|\mathcal{V}\| = \binom{n}{2} = \frac{n(n-1)}{2},$$

where $n = \|V\|$. It is convenient notation to use an uppercase letter to represent a graph. So we can define a graph as $G = (V, E)$ or even $G(V, E)$, and for later reference we might just write graph G.

Let's take an example. Let $V = \{0, 1, 2, 3, 4\}$. Then

$$\mathcal{V} = \{\{0,1\},\{0,2\},\{0,3\},\{0,4\},\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,4\},\{3,4\}\}.$$

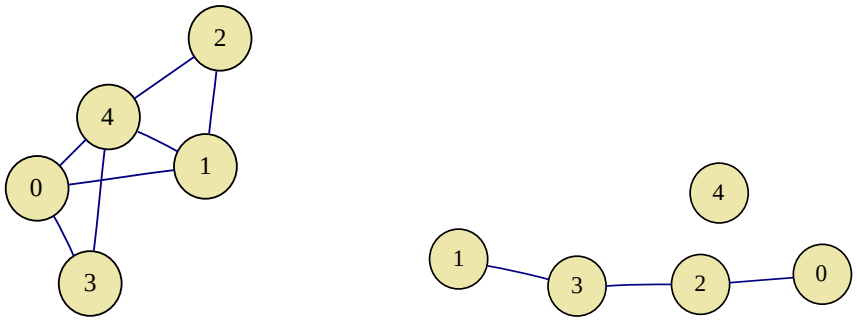

Figure 4.3: Left: a simple graph with 5 nodes and 7 edges. Right: its complement.

Let $E = \{\{0,1\},\{0,3\},\{0,4\},\{1,2\},\{1,4\},\{2,4\},\{3,4\}\}$. Now we can define the graph $G(V,E)$, whose graphic representation is shown in Figure 4.3.

The labels of the nodes, are not important, in the sense that the same exact graph, or "shape" can be drawn using another set of edges. What is important however is the topology of the graph. Nor is it important that we chose natural numbers for the labels. As in the definition, $\mathcal{V}$ can be any set, even not a subset of the real numbers. Here are some more definitions that will be useful further on.

Definition 22. Let $G = (V,E) \in \mathcal{G}$. Define the graph $\widetilde{G} = (V,\widetilde{E})$, where $\widetilde{E} = \mathcal{V} \setminus E$. Then $\widetilde{G}$ is called the *complement* of G.

Definition 23. Let $G(V,E) \in \mathcal{G}$. If $\omega(F) = \chi(F)$ for every $F \subseteq G$, then we say G is *perfect*.

4.2.2 Two NP-complete problems

There are two problems related with graphs that we are interested in. Both of them are NP-complete, which means that they cannot be solved in polynomial time. Following the notation given above we give the following.

Definition 24. A graph $G(V,E)$ is *complete*, if $E = \mathcal{V}$. A subgraph F of G is called a *clique* if it is complete. The number of vertices in a largest clique of G is called the *clique number* of G. The minimum number of different labels that can be assigned to each node of G, such that every label appears in each clique at most once, is called the *chromatic number* of G.

We will denote the clique number of a graph G by $\omega(G)$ and its chromatic number by $\chi(G)$. As we've already mentioned, calculating ω and χ for an arbitrary graph, are two very complex problems. However, in special cases, there is a way to compute both those numbers in polynomial time.

4.2.3 The sandwich theorem

In 1981, Grötschel, Lovász, and Schrijver [Knuth, 1994] proved that for any graph G, there is a real number that lies between $\omega(G)$ and $\chi(G)$, which can be calculated in polynomial time. The computation involves solving a semidefinite program for the complement of the graph and that makes it a convex optimization problem.

Definition 25 (The Lovász ϑ function)**.** Let $G(V, E) \in \mathcal{G}$. Consider the following optimization problem.

$$\begin{array}{ll} \text{maximize} & \mathbf{trace}\, \mathbb{1}_n \mathbb{1}_n^T X \\ \text{subject to} & x_{ij} = 0 \text{ for } \{i,j\} \in E \\ & \mathbf{trace}\, X = 1 \\ & X \succcurlyeq 0, \end{array} \tag{4.3}$$

where $X = [x_{ij}] \in \mathbb{R}^{n \times n}$, $\mathbb{1}_n$ is the n-dimensional column vector of all ones and $n = \|V\|$. Define $\vartheta : \mathcal{G} \to \mathbb{R}$ such that $\vartheta(G) = f^*$, where f^* is the optimal value of (4.3).

As given, (4.3) is not an SDP. But we can construct an equivalent SDP in standard form. First we note that $\mathbf{trace}\, \mathbb{1}_n \mathbb{1}_n^T X$ is just another way of writing $\sum_{i,j} x_{ij}$. We can now force the $\mathbf{trace}\, X = 1$ constraint, by demanding

$$x_{n-1,n-1} = 1 - \sum_{0 \leqslant i < n-1} x_{ii}.$$

We can also enforce the $x_{ij} = 0$ for $\{i, j\} \in E$ constraint a priori. Then our objective is simplified, as all elements of the diagonal will add to 1 and the result will be

$$\sum_{i,j} x_{ij} = 1 + 2 \sum_{i,j \mid \{i,j\} \notin E \text{ and } i \neq j} x_{ij}.$$

And this is exactly the quantity we want to maximize. The factor of two will appear, due to symmetry($\{i, j\} \in E \Leftrightarrow \{j, i\} \in E$). Our problem now, would be maximizing the above quantity, under the condition that the original matrix stays positive semidefinite. Let $m = \|E\|$ and $d = \|\mathcal{V}\| -$

```
'''
Calculation of theta(G), for
G=({0,1,2,3,4} , {{0,1},{0,3},{0,4},{1,2},{1,4},{2,4},{3,4}})
'''
from cvxopt.solvers import sdp
from cvxopt.base import matrix,spmatrix

def thetasmpl(n,m,ce):
  'Given the edges of the complement of G, returns theta(G)'
  d=n*(n-1)/2-m+n
  c=-matrix([0.0]*(n-1)+[2.0]*(d-n))
  Xrow=[i*(1+n) for i in xrange(n-1)]+[a[1]+a[0]*n for a in ce]
  Xcol=range(n-1)+range(d-1)[n-1:]
  X=spmatrix(1.0,Xrow,Xcol,(n*n,d-1))
  for i in xrange(n-1): X[n*n-1,i]=-1.0
  sol=sdp(c,Gs=[-X],hs=[-matrix([0.0]*(n*n-1)+[-1.0],(n,n))])
  return 1.0+matrix(-c,(1,d-1))*sol['x']

if __name__=='__main__':
  M,N=7,5 # the number of edges and nodes in G
  cedges=[(0,2),(1,3),(2,3)] # list of the edges of the complement of G
  print thetasmpl(n=N,m=M,ce=cedges)
```

Program 4.2: *graphex0opt.py*. The program returns 2, which is the clique number and the chromatic number of the complement of G.

$m+n$. This is the number of non-zero elements in X in the lower triangular (which is exactly $\|\widetilde{E}\|$) plus the elements of the diagonal. If we subtract one more element, which will be x_{n-1n-1}, since we made that redundant, we have the dimension of our problem. That means that if our optimization variable is x, then $x = [x_{k^{-1}(0)}, x_{k^{-1}(1)}, \ldots, x_{k^{-1}(d-1)}]^T \in \mathbb{R}^{d-1}$, where k is any isomorphism

$$k : \widetilde{E} \cup \{\{0,0\},\{1,1\},\ldots,\{n-2.n-2\}\} \rightarrow \{0,1,2,\ldots,d-1\}.$$

Now let c be a $(d-1)$-dimensional vector. That will be our multiplication vector in our objective. To make $c^T x$ give the result we want, we put zeros in $n-1$ places in c and twos in the rest. The positions of the zeros will correspond to the positions of the diagonal elements of X in x. We have to leave the constant outside for now, but we can add it in the end. The last thing to do, is to construct the constraint $X \succcurlyeq 0$. What is left now is to place the elements of x back in their original place in X. Of course, in order to retain the standard form, we will have the constraint $-X \preccurlyeq 0$

instead. So now we have the following problem

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & -\sum\limits_{0 \leqslant i < d-1} x_{k^{-1}(i)} H_i - H_{d-1} \preccurlyeq 0, \end{array} \tag{4.4}$$

where $H_i \in \mathbb{R}^{n \times n}, 0 \leqslant i < d-1$ are matrices of all zeros except for the element in position $(k^{-1}(i))$ which will be 1. Also, if $c_i = 0$ then the matrix H_i will have a -1 in the last position $(n-1, n-1)$. Finally, $H_{d-1} \in \mathbb{R}^{n \times n}$ will be a matrix of all zeros, except for the element in the last position $(n-1, n-1)$, which will be 1. Here's an illustration of the problem's parameters, where we place the diagonal elements of X in the first $n-1$ positions of x.

$$X = \begin{bmatrix} x_{00} & x_{01} & & \cdots \\ x_{10} & \ddots & & \\ \vdots & & 1 - \sum\limits_{0 \leqslant i,j < n-1} x_{ij} & \end{bmatrix},$$

$$c = [0, \ldots, 0, 2, \ldots, 2]^T \in \mathbb{R}^{d-1}, c_i = 0 \Leftrightarrow i \in \{0, 1, \ldots, n-1\},$$

$$x = [x_{00}, x_{11}, \ldots, x_{n-2n-2}, \ldots, x_{k(i)}, \ldots]^T, i \in \{n, n+1, \ldots, d-1\},$$

$$H_s = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & 1 & \vdots \\ 0 & \ldots & -1 \end{bmatrix}, \quad H_r = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & 1 & \vdots \\ 0 & \ldots & 0 \end{bmatrix}, \quad H_{d-1} = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & 1 \end{bmatrix}.$$

where $0 \leqslant s < n$, $n \leqslant r < d-1$ and the 1 in H_s and H_s goes in the $k(s)^{\text{th}}$ and the $k(r)^{\text{th}}$ position accordingly. It can be seen that (4.4), is a SDPin standard form, without the equality constraints. And if f^* is its optimal value, then $\vartheta(G) = f^* + 1$. Program 4.2 gives a function to solves (4.4) for this example in a very simple but adequate way for now. (A better function has been constructed to solve the problem and extract the data for the results furtherdown.)

Theorem 8 (The sandwich theorem)**.** *Let $G(V, E) \in \mathcal{G}$ and let $\widetilde{G}(V, \widetilde{E})$ be its* complement*. Then*

$$\omega(\widetilde{G}) \leqslant \vartheta(G) \leqslant \chi(\widetilde{G}). \tag{4.5}$$

(*Proof* in Lovász [1979].) An immediate result of this theorem, is that for perfect graphs, χ and ω can be calculated in polynomial time, as the ϑ function of the complement graph. But even for the general case, having a lower bound for the chromatic number and an upper bound for the clique number for any graph, is a big advantage. For historical reasons, here is the original definition of ϑ as given in Lovász [1979].

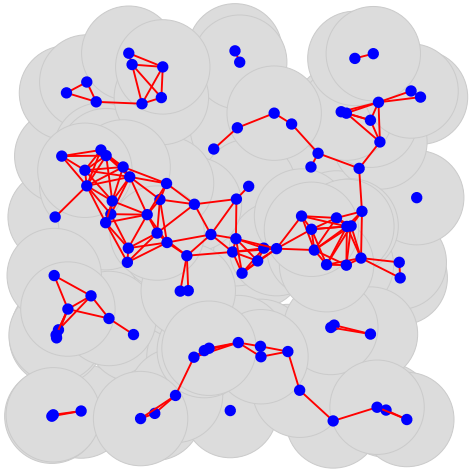

Figure 4.4: A geometric random graph with 100 nodes.[*.pdf picture generated with* generate_grg_graph.py*, written by* Dr. Keith M. Briggs.]

Definition 26. Let $G(V,E) \in \mathcal{G}$. Also let

$$\mathcal{A} := \{A \in \mathbb{S}^n \mid a_{ij} = 0 \text{ if } \{i,j\} \in E\}.$$

Denote $\lambda_0(A) \geqslant \lambda_1(A) \geqslant \ldots \lambda_{n-1}(A)$, the eigenvalues of A. Then

$$\vartheta(G) = \max_{A \in \mathcal{A}} \left| 1 - \frac{\lambda_0(A)}{\lambda_{n-1}(A)} \right|,$$

where $n = \|V\|$.

4.2.4 Wireless Networks

We consider a wireless network with n transmitter-receiver devices in fixed arbitrary positions. We are not interested in all the technical details of the model, except for the fact that every device receives or transmits information over a fixed radio frequency, or channel. For simplicity, we assume that all devices have the same radius of transmission. This model can be represented as a graph $G(V,E)$, where $V = \{0,1,\ldots,n-1\}$ and $\{i,j\} \in E$ if and only if device i is within the transmission radius of device j (and vice versa of course). If we place n devices randomly on the plane and choose the edges as described, what we have is a *geometric random graph* (Figure 4.4). Every clique in the graph represents a set of devices that can intercommunicate. That means, that every transmission of a device within the clique, will be detected by any other device in the clique, that receives from the same channel.

In most cases, this detection is undesired and considered interference. Then we must assign a different transmission channel to every device in the same clique, to avoid interference. Even in cases when this detection is desired, we still may have to assign channels in such a way that every device in the same clique transmits to a different channel.

Under the above constraints, the problem of finding the least number of channels to be used in a wireless communications network is exactly the problem of defining the chromatic number χ of the graph G, that represents the network as described earlier. Then finding the ϑ number of the complement of G, will give us a lower bound on χ in polynomial time, as well as an upper bound on ω, the maximum number of devices in the same transmission radius.

4.2.5 Statistical results

We were interested in investigating the distribution of the ϑ function over a random sample of graphs with roughly the same number of edges. We choose the geometric random graph model, because it is a realistic model for wireless communications applications.

We took a sample of M geometric random graphs of n nodes and radius r. For every graph, the nodes where randomly placed in the unit square and we varied r from 0 to 1 with step 0.1. That meant that when $r = 0$, no nodes would be connected, and the result would always be an empty graph. When $r = 1$, for every pair of nodes there was a probability p that they would be connected with an edge, with $\pi/4 \leqslant p \leqslant 1$. To take one sample, we fixed n, r, generated the geometric random graph with those attributes, computed its ϑ and repeated the operation M times. We then plotted the result in a histogram. We varied r as described and thus generated 10 histograms, which we plotted on the same graph. The results for $M = 500, n = 30$ and $M = 100, n = 80$ are shown in Figure 4.5.

To interpret the result we take into consideration Theorem 8. In the case of zero radius, the sample is consistent of M empty graphs. The complement of the empty graph, is a complete graph. The χ and ω numbers of a complete graph are both equal to n, the number of nodes in the graph. This is simple to understand, as in a complete graph, the whole graph is a clique. Thus, the largest clique of the graph is the graph itself and the number of its nodes is $\omega = n$. And it is straightforward to see how $\chi = n$ as well. On the other hand, when $r = 1$ most of the graphs in the sample will be complete, or almost complete. That results in empty graph complements, which trivially have $\omega = \chi = 1$ or complements that only have

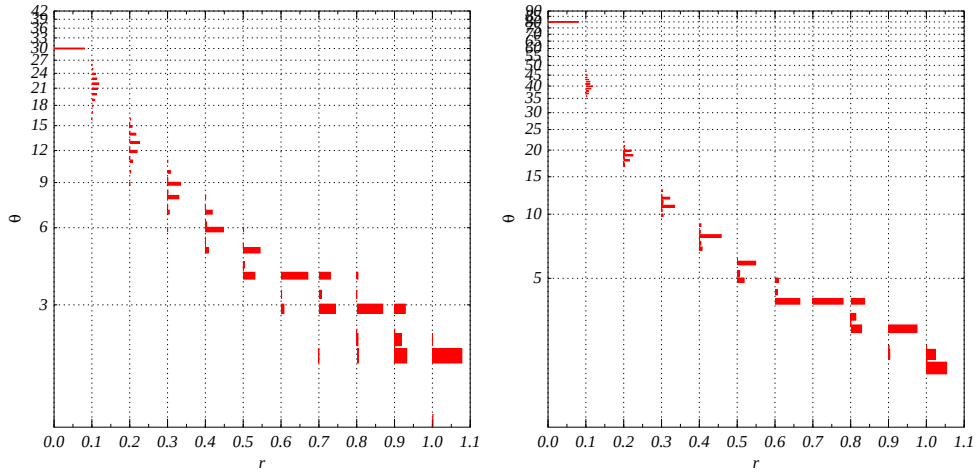

Figure 4.5: Distribution of the ϑ function over 10 samples of geometric random graphs of 30 (left) and 80 (right) nodes on a logarithmic scale.

one edge, or a few but isolated edges, in which case $\omega = \chi = 2$. For all the cases in between, the best information we can acquire for ω and χ is the Lovász ϑ function, whose distribution we have illustrated in Figure 4.5.

4.2.6 Further study

Except for geometric random graphs and the apparent immediate applications of that model, it is also very interesting to investigate the distribution of the ϑ function in a more theoretical level. For that purpose we will use a different random graph model, the *Erdős-Rényi* random graph. In short, $G(n, p)$. As implied by the notation, in this model every edge appears in the graph with probability p. For example, to generate a $G(n, 0.5)$, we would go through all possible pairs of the n nodes and every time would connect them with an edge, depending on the outcome of a coin toss.

We computed the distributions of $G(n = 50, 0 \leqslant p \leqslant 1)$ and plotted them as shown in Figure 4.6. First of all we must stress the fact that although ω and χ take integer values, there is nothing in the definition to imply that for ϑ, except for perfect graphs. However, as shown in Figure 4.6, the distribution of ϑ is strongly concentrated on integer values, with non-integer values occurring less often but almost certainly. For example, we know [Lovász, 1979] that for *cycle graphs* with m nodes and m odd, there is a closed formula. A cycle graph intuitively, is a graph that looks like a closed chain. More formally, if $C_m = G(V, E)$, where $V = \{0, \ldots, m-1\}$ is

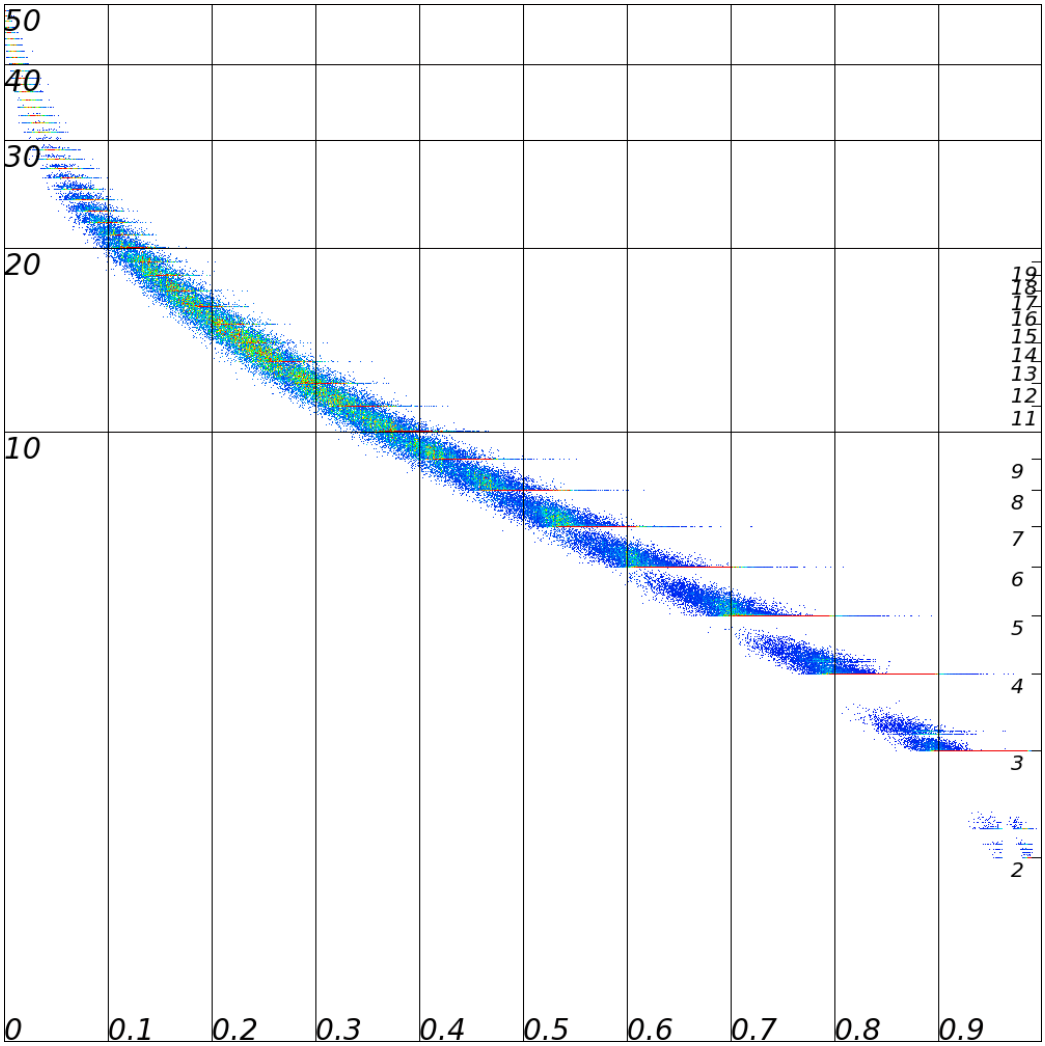

Figure 4.6: The distribution of ϑ over 1000 samples of 100 $G(n,p)$ graphs each of $n = 50$ and $0 \leqslant p \leqslant 1$. The horizontal axis is p and the vertical is the logarithm of $\vartheta(G)$. For every observed point a dot is drawn on the picture. At first, the dot is blue. Whenever a dot has to be drawn on top of an existing dot, the colour of the dot moves up the spectrum and closer to red. Thus, blue areas in the graph indicate the lowest frequency and red indicate the highest frequency.[*Data and .png picture acquired in collaboration with* Dr. Keith M. Briggs]

a cycle graph, then $E = \{\{0,1\},\{1,2\},\ldots,\{m-2,m-1\},\{0,m-1\}\}$. For these graphs then, if m is odd

$$\vartheta(C_m) = \frac{m\cos(\pi/m)}{1+\cos(\pi/m)}. \tag{4.6}$$

One hypothesis then would be that one case when non-integer ϑ occurs, is when the graph contains an m-cycle with m odd.

We could make more very interesting hypotheses. For instance, if one observes at the ϑ of small graphs in Figures 4.16 and 4.17, it seems that whenever a graph is consistent of smaller isolated subgraphs, the ϑ of the graph is equal to the sum of the ϑ of its isolated subgraphs. The investigation of these conjectures would make a very interesting research subject on its own and would much likely contribute to an even more efficient way of computing the ϑ function.

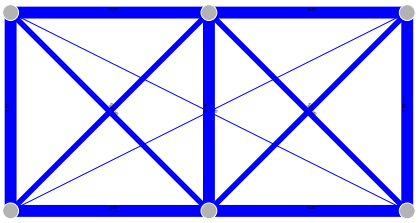

Figure 4.7: A very simple truss.

4.3 Optimal truss design

This example was inspired by *"Truss Design and Convex Optimization"* [Freund, 2004]. A truss is a structure mathematically similar to a graph. To be precise, the set of trusses, could be considered as a subset of the set of graphs. Bridges are trusses and so are cranes, the roofs on big structures, like train stations and football fields, and the Eiffel Tower. Since they are real constructions and not theoretical objects, they are either 2-dimensional or 3-dimensional. In this section, we are considering only 2-dimensional (rectangular) trusses. These examples are applicable mostly to the design of bridges, but other applications can be found as well.

4.3.1 Physical laws of the system

A truss, as a graph object, consists of nodes and edges. The positions of the nodes are significant, as are the dimensions of the edges. We will refer to the edges of a truss as *bars*, as it is consistent with the structural character of the model. We will enumerate the nodes and the bars and denote the length of bar k by L_k. Another important feature of the truss is the material we will use to construct it. To keep our model simpler and maybe more realistic, we suppose that all bars are made of the same material. The material the bars are made of determines the stiffness of the model. A measure of stiffness is *Young's modulus*, denoted E or Y and measured in units of pressure (**pascal**). Here is Young's modulus for different material, in GPa (Gigapascal) [data taken from Wikipedia [2008c]:

Rubber	0.01-0.1
Oak wood	11
Bronze	103-124
Titanium	105-120
Steel	190

Most nodes of the truss will be free to move, but naturally, some will be fixed in place, or static. When an external force F is applied on a free node, the truss will deform elastically and some free nodes will displace. That causes the bars to stretch or compress, which results on internal forces f_k from the bars to be applied on the nodes they are connecting. These forces counteract the external force. The equilibrium condition can be written as

$$Af = -F, \tag{4.7}$$

where f is the vector of all internal forces f_k and A is the *projection matrix*. The k^{th} column of A corresponds to bar k. The rows of A correspond to the horizontal, or vertical axis, taken once for every free node. Instead of enumerating the rows of A it would be easier to label them. So if our truss has 4 nodes and nodes $1, 3$ are static, then the rows of A will be labeled as $\{0_x, 0_y, 2_x, 2_y, 4_x, 4_y\}$. Now the $j_x i$ element of A will be the projection of bar i onto the horizontal axis, with j taken as the origin, if j is one of the nodes that bar i connects. Otherwise it will be zero. Likewise for the $j_y i$ element. If we denote u the vector of displacements of the free nodes, given projected on the x and y axis, and t the vector of the volumes of the bars, then the internal force f_k on bar k is given by

$$f_k = -\frac{E}{L_k^2} t_k (a_k)^T u, \tag{4.8}$$

where a_k is the k^{th} column of A. We note that $-(a_k)^T u$ is the first order Taylor approximation of the distortion of bar k, which is acceptable for such small displacements of the nodes. The elastic energy stored in the truss when under stress, which is the same as the work performed by the external force F that causes the stress on the truss is given by

$$\frac{1}{2} F^T u.$$

This quantity is also called the *compliance* of the truss.

4.3.2 The truss design optimization problem

Our objective is to construct the truss to be as stiff as possible. In other words make it resistant to external forces. Our problem then will be to find which vector of volumes t will minimize the compliance. We also have to impose an upper bound on the total volume of bars, otherwise the solution would be infinite volume for all bars. Such a constraint would

express the fact that there is either finite resources to construct the truss, or even that the cost of the construction must not exceed a certain amount. We can now write the problem as:

$$\begin{array}{ll} \text{minimize} & \dfrac{1}{2}F^T u \\ \text{subject to} & \left[\displaystyle\sum_{k=1}^{m} t_k \frac{E_k}{L_k^2}(a_k)(a_k)^T\right] u = F \\ & \\ & Mt \preccurlyeq d \\ & t \succcurlyeq 0. \end{array} \tag{4.9}$$

The first constraint is a result of the equilibrium conditions (4.7) and (4.8). The second is a linear equality constraint on the volumes of the bars, which can be the upper bound on the total volume as well as upper bounds on specific bars. The last constraint only dictates that volumes are positive quantities. This is equivalent (proof in Freund [2004] pages 37-40) with the following:

$$\begin{array}{ll} \text{minimize} & \theta \\ \text{subject to} & \begin{bmatrix} \theta & F^T \\ F & \left[\displaystyle\sum_{k=1}^{m} t_k \frac{E_k}{L_k^2}(a_k)(a_k)^T\right] \end{bmatrix} \succcurlyeq 0 \\ & \\ & Mt \preccurlyeq d \\ & t \succcurlyeq 0, \end{array} \tag{4.10}$$

which is an SDP. There is also a SOCP equivalent of the problem, but we prefer the SDP for the reasons described in the end of Chapter 3. To illustrate the SDP form of the problem above better and to be consistent with our definitions, we rewrite the problem as

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & \mathbf{diag}(Mt - d, -t, -Q) \preccurlyeq 0, \end{array} \tag{4.11}$$

where

$$Q = \begin{bmatrix} \theta & F^T \\ F & \left[\displaystyle\sum_{k=1}^{m} t_k \frac{E_k}{L_k^2}(a_k)(a_k)^T\right] \end{bmatrix},$$

$c = [1, 0, \ldots, 0]^T$, $x = [\theta, t_0, \ldots, t_m]^T$ and $\mathbf{diag}(A_0, \ldots, A_s)$ is the block diagonal matrix with blocks $A_0, \ldots, A_s$.

4.3.3 Results

We have developed a python module (given in the Appendix) for **CVXOPT** to solve this problem. Some interesting results are shown in Figures 4.9-4.14. Of course our results are weak, in the sense that we don't consider variables like construction difficulties, nor do we have the actual data to know what material would be used or if we should take other factors into account like seismic resistance, not to mention that the problems we solved were planar. The models below, show the solutions to the problem from a mathematical point of view. However, such a theoretical model can prove useful, as it can produce ideal designs, that may not be constructible, but into which additional constraints and realistic parameters can be put, resulting to constructible solutions that are as close to the theoretical optimal solution as possible.

The simplest Figure (4.9), resembles the kingpost design. The design was used in Medieval, Queen Anne and Gothic Revival Architecture as roof support and has also been used in early aviation on wire-based aircrafts.

In Figure 4.13, we have the model for the Tyne bridge in Newcastle-Upon-Tyne. The bridge is supported on the ground, but the road is higher, close to the middle of the bridge. The actual bridge has a double arch on top, whereas our model suggests that the optimal design should have three or four arches. While experimenting with truss examples, the same pattern seemed to arise every time. Instead of the triangulation we are used to see in most trusses, like the Eiffel tower, or all cranes (Figure 4.11), the model tends to add extra thin layers next to layers that sustain most of the stress. A characteristic example is displayed in Figure 4.14.

Another interesting pattern, is the shape of the arch that seems to occur naturally in the design of bridges. We took the bridge model shown in Figure 4.14 and tried to fit the points on the main arch in different models. The data seemed to fit almost perfectly to the *catenary*. That is a curve of the form $\alpha+\beta\cosh\gamma x$ and represents the shape of a hanging chain, or rope, when supported at its ends.

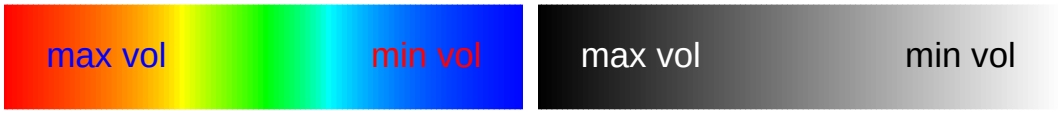

Figure 4.8: The legend for the figures below (left: in color, right: in gray-scale). Although line thickness is proportional to the actual thickness of the bars, color labeling is also used for the scaling volumes. A shade closer to red (black in gray-scale), means larger volume than a shade closer to blue (white). These figures are zoomable pdf pictures. In quadruple zoom level or above, one can see the actual volumes on the bars (in arbitrary units). Nodes are drawn as small gray disks. Bigger black disks denote static nodes. Forces are drawn with black arrows.

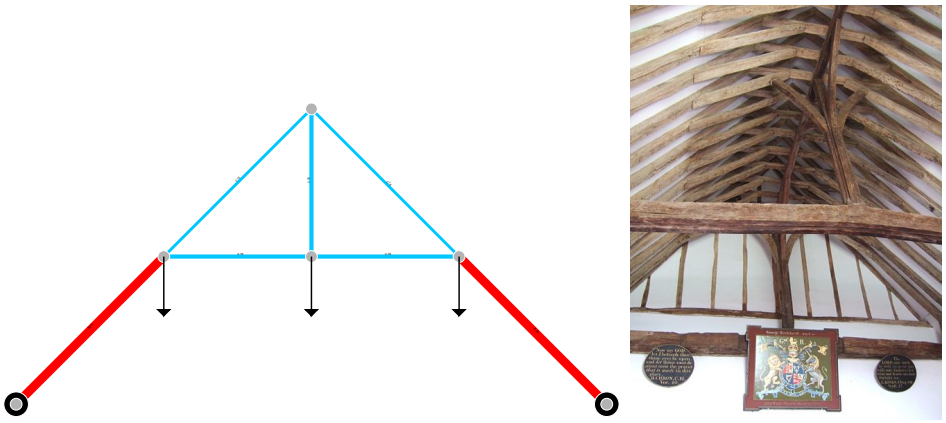

Figure 4.9: The king post design. Photo taken from Wikipedia [2008a]

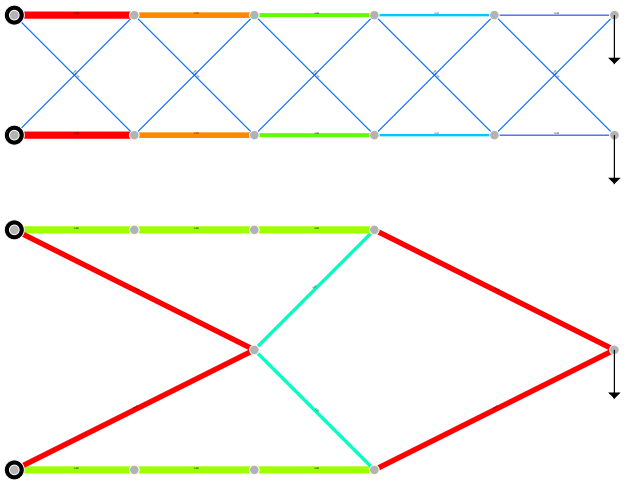

Figure 4.10: Optimal hanging sign designs.

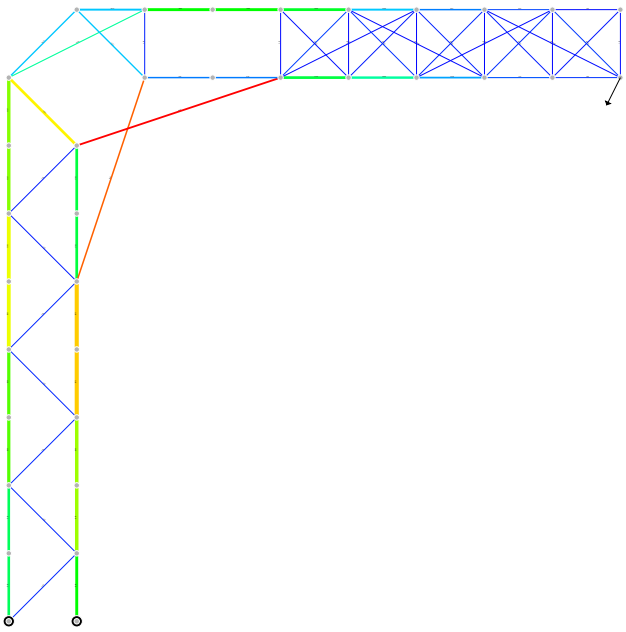

Figure 4.11: Optimal design for a simple crane.

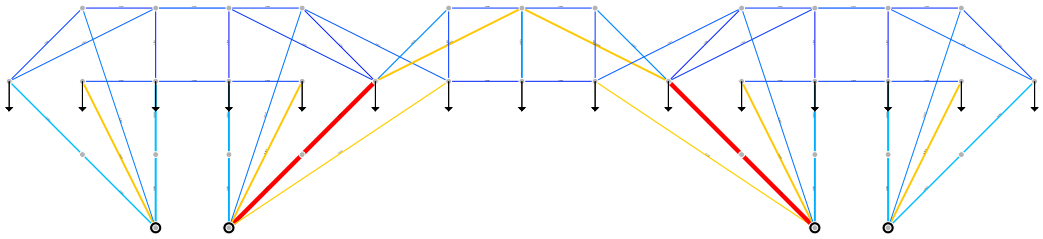

Figure 4.12: Optimal design for a train bridge.

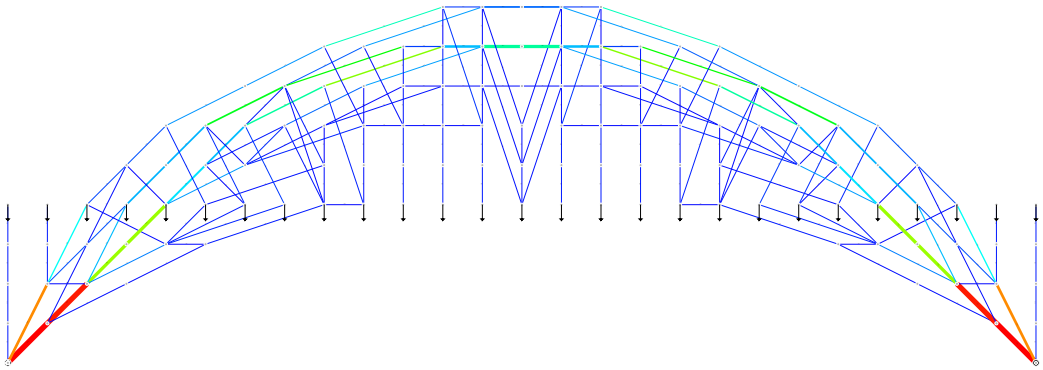

Figure 4.13: The Tyne bridge model. Photo taken from Wikipedia [2008b]

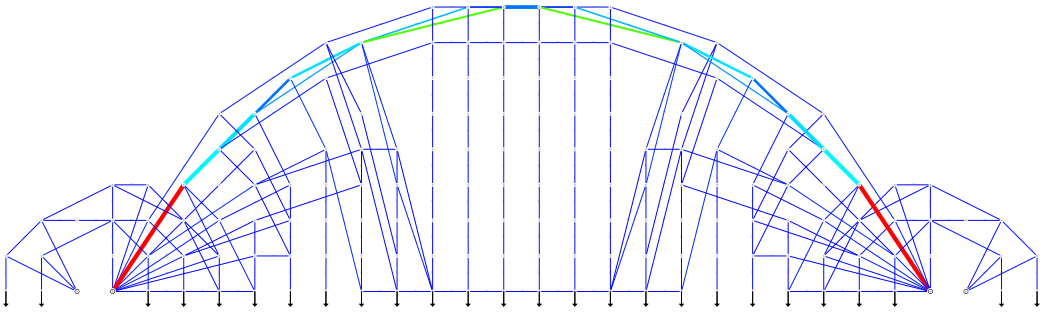

Figure 4.14: Optimal design for a road bridge.

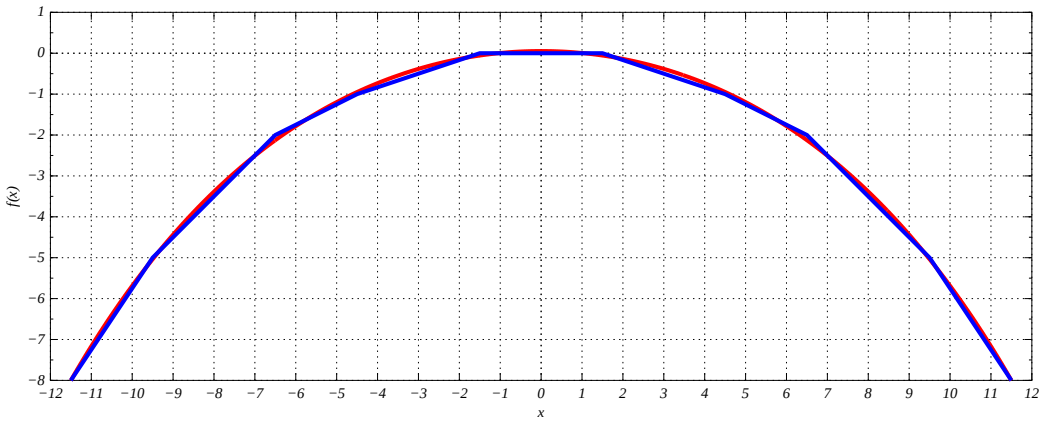

Figure 4.15: Fitting of the arch of the model in Figure 4.14. The blue line represents the arch and the red line is the graph of $f(x) = \alpha - \beta \cosh(\gamma x)$. For this drawing the parameter values $\alpha = 4.2, \beta = 4.1, \gamma = 0.15$ were used. However in the fitting process, the standard errors came out around $1, 1, 0.01$ accordingly, which means that the data could be fitted by a wide range of catenary curves.

4.4 Proposed research

4.4.1 Power allocation in wireless networks

We consider a problem taken from the field of wireless telecommunications [Boyd and Vandenberghe, 2004, page 196]. Consider n pairs of transmitters (T^x) and receivers (R^x), located in different positions in a closed area. To be realistic, we must think that every transmitter is closer to its pair receiver, than to any other receiver. Every transmitter T_i^x is supplied with power p_i. The signal power at R_j^x from T_i^x will be proportional to p_i and inversely proportional to some power of the distance R_j^x and T_i^x. Of course at R_i^x, the only desirable signal is from T_i^x. The sum of all signal powers from all T_j^x, $j \neq i$ is considered *interference*.

Let $G \in \mathbb{R}^{n \times n}$ be the matrix of all *path gains* from every transmitter to every receiver. In short, that means that $G_{ij}p_i$ will be the signal power from T_i^x at R_j^x. Let C be the matrix whose non-zero elements are on the diagonal and equal to the diagonal elements of G. Let also $R = G - C$. Obviously, I will differ with G only on the diagonal, the elements of which will be all zeros for R. Now let $S = Cp$ and $I = Rp$, where $p = [p_0, \ldots, p_{n-1}]^T$ and $S, I \in \mathbb{R}^n$. Then the (desired) signal power at R_i^x will be S_i and the interference at R_i^x will be I_i.

The objective here will be to have the best possible signal power with the least possible interference at every receiver. Since we take the position of the devices to be fixed, we can only vary p, which will be the optimization variable. If we also add noise σ to our model, the objective will be to maximize

$$\min_{0 \leqslant i < n} \frac{S_i}{I_i + \sigma_i}$$

which is the minimum *signal to interference plus noise ratio* (SINR) over all receivers. There will be upper bounds $p^{\max} \in \mathbb{R}^n$ on the maximum power to be allocated to every receiver (either due to legal restrictions or electrical capacity of the device), and also upper bounds $p^R \in \mathbb{R}^n$ on the total received power at every receiver. Now we can write the problem as

$$\begin{array}{ll} \text{minimize} & \max\limits_{0 \leqslant i < n} \dfrac{-S_i}{I_i + \sigma_i} \\ \text{subject to} & p \preccurlyeq p^{\max} \\ & Gp \preccurlyeq p^R \end{array} \tag{4.12}$$

where we have used the obvious equivalence of minimizing over the maximum of $-$SINR to maximizing over the minimum of SINR for the sake of

consistency. Both inequalities are componentwise, as already discussed in section 2.4.

This problem is an example of generalised LFP and as such, it is quasi-convex. We have tried solving it as described in section 3.1, but the solvers we used seem to fail. We also tried different approaches than our python tools, but we were unable to find a complete package designed to cope with this kind of problems. The difficulty is that the objective is not a smooth function. Discontinuities in the gradient, make the solvers we used produce irrelevant results, due to the fact that they rely on the gradient methods which as it seems require a certain degree of smoothness.

Appendix

truss_optimize_05.py A truss optimal design module for **CVXOPT**.

```
#!/usr/bin/python
# optimize a truss design using semi-definite programming
# Costas Skarakis 2008-07-23
# theory from "Truss Design and Convex Optimization" by Robert M. Freund

from sys   import argv,exit
from math  import pi,cos,sin,hypot,atan2
from time  import time
from array import array
import random
import cPickle

try:
  from cvxopt.base    import matrix,spmatrix,spdiag,sparse
  from cvxopt.solvers import sdp
except:
  print 'cvxopt not found - please install from http://abel.ee.ucla.edu/cvxopt'
  exit(1)

try:
  from reportlab.pdfgen import canvas
  from reportlab.lib.colors import gray,black
  got_reportlab=True
except:
  print 'ReportLab not found - please install from http://www.reportlab.org/'
  got_reportlab=False

try:
  from hue import h2tuple
except:
  print 'No module hue - using grayscale'
  def h2tuple(h):
    return (h,)*3

pictscale=33
costpervolunit=1.0

class Graph:
  'A graph class'
  def __init__(s,E=[],n=1):
    s.d=None
    E=flat(E)
    assert False not in [type(i)==int for i in E]
    assert len(E)%2==0
    assert True not in [E[i]==E[i+1] for i in range(len(E))[::2]]
```

```
  if not E:
    s.V=range(n)
  else:
    s.V=range(max(E+[0])+1)
  s.E=(array('i',list(E[::2])),array('i',list(E[1::2])))
  s.m=len(s.E[0])
  s.n=len(s.V)

def __neg__(s): # complement(graph)
  if not s.d:
    s.d=dict([(i,[]) for i in s.V]) #range(s.n)])
    for i,j in zip(s.E[0],s.E[1]):
      s.d[i].append(j)
      s.d[j].append(i)
  c=Graph(n=s.n)
  l=(s.n*(s.n-1))/2-s.m
  a0=array('i',[0]*l)
  a1=array('i',[0]*l)
  count=0
  for i in s.V:#range(s.n):
    for j in s.V[i+1:]:#range(i+1,s.n): # i<j<n
      if not j in s.d[i]:
        a0[count]=j
        a1[count]=i
        count+=1
  c.E=(a0,a1)
  c.m=l
  c.n=s.n
  c.d=None
  return c

def __str__(s):
  r=''
  if s.V==[0]: return r
  for i in zip(s.E[0],s.E[1]):
    r+=repr(i[0])+'--'+repr(i[1])+' '
  for i in s.V:
    if i not in s.E[0] and i not in s.E[1]: r+=repr(i)+' '
  r+='\n'
  return r[:-1]

def __repr__(s):
  r='Graph(['
  for k,l in zip(s.E[0],s.E[1]):
    r+='('+repr(k)+','+repr(l)+')'+','
  if r[-1]==',':
    r=r[:-1]
  r+='])\n'
  return r[:-1]

def __add__(s,o):
  o=flat(o)
  assert False not in [type(i)==int for i in o]
  assert len(o)%2==0
  assert True not in [o[i]==o[i+1] for i in range(len(o))[::2]]
  s.m+=len(o)/2
  d=len(o)/2
  c=0
  for a in o:
    if a not in s.V: c+=1
  s.V+=[0]*c
  c=0
```

```
    for a in o:
      if a not in s.V:
        c+=1
        s.V[s.n+c-1]=a
    s.V.sort()
    s.n=len(s.V)
    for r in o[::2]:
      s.E[0].append(r)
    for r in o[1::2]:
      s.E[1].append(r)
    return s

  def __iter__(s):
    return iter(zip(s.E[0],s.E[1]))

  def __len__(s):
    return len(s.E)/2

  def add_node(s,b):  # b is redundant
    s.n+=1
    s.V=range(n)
    s.d=None

  def add_nodes_from(s,nbunch):
    b=flat(nbunch)
    s.n+=len(b)-1
    s.V=range(s.n)
    s.d=None

  def add_edges_from(s,ebunch):
    b=flat(ebunch)
    s.__add__(b)

  def add_path(s,nlist):
    k=len(nlist)-1
    l=[(-1,-1)]*k
    for i in range(k): l[i]=(nlist[i],nlist[i+1])
    l.sort()
    s.add_edges_from(l)

  def delete_node(s,n):
    if n in s.V:
      s.n=s.n-1
      s.V=range(s.n)

  def order(s):
    return s.n

  def size(s):
    return s.m

  def edges(s):
    l=[(s.E[0][i],s.E[1][i])for i in range(s.m)]
    return l

  def nodes(s):
    return s.V

### end of class Graph ###

class Truss(Graph):
  ' A truss class '
```

```
def __init__(s,youngs_modulus=2e6,Fd={},lnth=0,hght=0,dist=1,st=[],maxuv=0.8):
  Graph.__init__(s)
  s.dist=dist
  s.bunch=[]          # bunch of bars with volumes with upper bounds
  s.bd=[]             # the upper bounds
  if lnth and hght: # make rectangular truss
    if not dist<=max(lnth,hght):
      print '\nWarning: dist too large.\n'
    def barlength(x,y):
      return hypot(x/hght-y/hght,x%hght-y%hght)
    def allow(x,y):
      xj,yj=x%hght,y%hght
      xi,yi=x/hght,y/hght
      a=(xj!=yj)
      b=(xi!=yi)
      c=((xj-yj)!=(xi-yi)) and ((xj-yj)!=(yi-xi))
      d=(min(lnth,hght)>2) or c
      e=(barlength(x,y)<=hypot(1,dist))
      f=abs(barlength(x,y)-hypot(1,1))<1e-14
      return a and b and c and e or f
    h=hght+1
    l=lnth+1
    s.st=st
    s.lnth=lnth
    s.hght=hght
    s.add_nodes_from(range(lnth*hght))
    nd=s.nodes()
    nd.sort()
    s.fr=[i for i in nd if i not in s.st]
    s.fr.sort()
    for k in range(lnth): s.add_path(range(hght*k,hght*(k+1)))
    for k in range(hght): s.add_path(nd[k::hght])
    s.add_edges_from([(x,y) for x in nd for y in nd[x:] if allow(x,y)])
    s.ls=s.edges()[:]
    s.ls.sort()
    s.L=dict([((i,j),barlength(i,j)) for i,j in s.ls])
    s.A=dict([(e,dict([((i,t),0.0) for i in s.fr for t in ('x','y')])) for e in s.ls])
    for i,j in s.A: # create the projection matrix
      sintheta=(j%hght-i%hght)/s.L[(i,j)]
      costheta=(j/hght-i/hght)/s.L[(i,j)]
      if i in s.fr:
        s.A[(i,j)][i,'y']=sintheta
        s.A[(i,j)][i,'x']=costheta
      if j in s.fr:
        s.A[(i,j)][j,'y']=-sintheta
        s.A[(i,j)][j,'x']=-costheta
    keys,keysdeep=s.A.keys(),s.A[(0,1)].keys()
    keys.sort()
    keysdeep.sort()
    s.A=[[s.A[e][i] for i in keysdeep] for e in keys]
  else:
    s.lnth=lnth
    s.hght=hght
    s.st=st         # the static nodes of the truss
    s.fr=[]         # the free nodes of the truss
    s.ls=[]         # the list of bars lexicographically sorted
    s.A=matrix([])
    s.L={}
  s.Y=[youngs_modulus]*s.size() # stiffness of the bars
  s.maxunitvol=maxuv # maximum volume of a bar
  s.cost=maxuv*costpervolunit*s.size()          # maximum cost
  s.Fd=dict([((i,t),0.0) for i in s.fr for t in ('x','y')])
```

```
    for i in Fd:
      s.Fd[i]=Fd[i]
    s.optd=[maxuv]*s.size()     # optimal design for bar volumes

  def __repr__(s):
    r='Truss(youngs_modulus='
    if s.Y:
      r+=s.Y[0]
    else:
      r+='2e6'
    r+=',Fd='+repr(s.Fd)+',lnth='+repr(s.lnth)+',hght='+repr(s.hght)+',dist='
    r+=repr(s.dist)+',st='+repr(s.st)+',maxuv='+repr(s.maxunitvol)+')\n'
    return r[:-1]

  def add_forces(s,Fd):
    for i in Fd:
      s.Fd[i]=Fd[i]

  def bars(s):
    l=s.edges()
    return l

  def add_overall_cost_constraint(s,cost):
    s.cost=cost

  def add_edge_constraints(s,ebunch,upbd):
    if not len(ebunch)==len(upbd):
      print 'Constraint vectors of different sizes ignored.'
    s.bunch+=ebunch
    s.bd+=upbd

  def clear_constraints(s):
    s.bunch=[]
    s.bd=[]
    s.cost=s.maxunitvol*costpervolunit*s.size()

  def solve(s):
    s.optd=minimum_compliance_sdp(s,s.Y,s.Fd,s.bunch,s.bd,s.cost)

  def draw(s,pdf_fn='truss.pdf'):
    bridgeview(s,s.optd,s.Fd,pdf_fn=pdf_fn)

### end of class Truss ###

class Easytruss(Truss):
  def __init__(s,youngs_modulus=2e6,Fd={},lnth=0,hght=0,dist=1,st=[],maxuv=0.8):
    s.cFd=Fd
    s.cst=st
    F,sta={},[]
    for k in st:
      if type(k)==tuple:
        sta.append(drooc(k[0],k[1],hght))
      else:
        sta.append(k)
    for k in Fd.keys():
      if type(k[0])==tuple:
        F[(drooc(k[0][0],k[0][1],hght),k[1])]=Fd[k]
      else:
        F[k]=Fd[k]
    Truss.__init__(s,youngs_modulus,F,lnth,hght,dist,sta,maxuv)
    if s.edges():
      s.cnodes=dict([(coord(a,s.hght),a) for a in s.nodes()])
```

```
        s.cedges=dict([((coord(a,s.hght),coord(b,s.hght)),(a,b))for a,b in s.edges()])

    def add_forces(s,Fd):
        F={}
        for k in Fd.keys():
            if type(k[0])==tuple:
                F[(drooc(k[0][0],k[0][1],s.hght),k[1])]=Fd[k]
            else:
                F[k]=Fd[k]
        Truss.add_forces(s,F)

    def cbars(s):
        l=s.cedges
        return l

    def add_edge_constraints(s,cebunch,upbd):
        s.cbunch=cebunch
        ebunch=[(-1,-1)]*len(cebunch)
        for i in range(len(cebunch)):
            t=[-1,-1]
            for j in (0,1):
                if type(cebunch[i][j])==tuple:
                    t[j]=drooc(cebunch[i][j][0],cebunch[i][j][1],s.hght)
                else:
                    t[j]=cebunch[i][j]
            ebunch[i]=(t[0],t[1])
        Truss.add_edge_constraints(s,ebunch,upbd)

    def solve(s):
        s.optd=minimum_compliance_sdp(s,s.Y,s.Fd,s.bunch,s.bd,s.cost)

    def draw(s,pdf_fn='truss.pdf'):
        bridgeview(s,s.optd,s.Fd,pdf_fn=pdf_fn)

### end of class Easytruss ###

def flat(seq,first=1):
    ' Removes depth levels from a sequence '
    l=[]
    for t in seq:
        try:
            t[0]
            l+=flat(t,0)
        except:
            l.append(t)
    return l

def minimum_compliance_sdp(T,Y,F,bunch=[],bd=[],cost=0.8):
    '''
    Takes a truss object T, a material stiffness list Y, a force
    list F and solves the optimal truss design semidefinite
    problem. The maximum cost contraint is cost.
    Upper bounds on the volumes of certain bars listed in bunch
    can be added in bd.
    '''
    k=F.keys()
    k.sort()
    F=[F[i] for i in k]
    t0=time()
    m,n=T.size(),2*len(T.fr)
    print 'Number of variables: %g'%(m+1,)
    L=[T.L[i] for i in T.ls]
```

```
  pt,ro,co,pr,rw,cl=[],[],[],[],[],[]
  for i in range(2*len(T.fr)):
    for j in range(m):
      if T.A[j][i]:
        ro.append(i)
        co.append(j)
        pt.append(T.A[j][i])
  A=spmatrix(pt,ro,co)
  Q=[(Y[k]/L[k]**2)*(A[:,k]*A[:,k].trans()) for k in range(m)]
  c=matrix([1.0]+[0.0]*m)
  if bunch and len(bunch)==len(bd):
    pr=[1.0]*len(bunch)
    rw,cl=range(m+2,m+2+len(bunch)),[1+edgeno(T,i,j) for i,j in bunch]
  else:
    bunch,bd=[],[]
  Gl=spmatrix([-1.0]*m+[1.0]*m+pr,range(1,m+1)+[m+1]*m+rw,2*range(1,m+1)+cl,(m+2+len(rw),m+1))
  hl=matrix([0.0]+[0.0]*m+[cost]+bd)
  forth=spmatrix(-1.0,[0],[0],((n+1)*(n+1),1))
  fort=[-spdiag([0,q]) for q in Q]
  roc,cor=[0]*n,range(1,n+1)
  forF=-spmatrix(F+F,roc+cor,cor+roc)
  Gs=[sparse([[forth]]+[[f[::]] for f in fort])]
  hs=[matrix(forF)]
  print 'setup time: %f'%(time()-t0,)
  t0=time()
  try:
    sol=sdp(c=c,Gl=Gl,hl=hl,Gs=Gs,hs=hs,solver='dsdp')
  except:
    sol=sdp(c=c,Gl=Gl,hl=hl,Gs=Gs,hs=hs)
  print 'solve time: %f'%(time()-t0,)
  return sol['x'][1::]

def edgeno(B,i,j):
  '''
  Given a truss B and an edge (i,j), finds the number
  of the edge, or prints an error message.
  '''
  r,t=-1,-1
  for k in range(B.size()):
    if B.ls[k]==(i,j):
      t=k
      r+=1
  if t<0: print i,j,'not an edge'
  if r>0: print i,j,'multiple edge'
  return t

def bridgeview(B,v,Fd,eps=9e-4,save=True,filename='_truss_optimize_last.data',pdf_fn='truss.pdf'):
  '''
  Takes a bridge object and a vector of weights and creates
  a pdf picture of the bridge with the thickness of the
  lines representing the bars, proportional to the thickness of the bars.
  The bars are coloured from blue to red, from the bar with the
  minimum volume up to the bar with the maximum volume.
  If save=True, the data from the last picture
  are saved in filename and the picture can be redrawn using
  drawlast(filename).
  '''
  if not got_reportlab: return
  psc=pictscale
  ht,ln=B.hght,B.lnth
  if save:
    f=open(filename,'w')
```

```
  cPickle.dump((B.bars(),v,Fd,B.L,ln,ht,B.st,B.fr),f)
  f.write('\n')
  f.close()
u=v[:]
v=[v[edgeno(B,i,j)]/B.L[(i,j)] for i,j in B.ls]
re,gr,bl=h2tuple(0);
re,gr,bl=h2tuple(1);
c=canvas.Canvas(pdf_fn,(psc*ln,psc*ht))
coo=[(i/ht,i%ht,j/ht,j%ht,v[edgeno(B,i,j)],u[edgeno(B,i,j)],i,j) for i,j in B.ls]
m,M=min(v),max(v)
mc,Mc=min(u),max(u)
n,N=min([abs(Fd[i]) for i in Fd]),max([abs(Fd[i]) for i in Fd])
scale=1.0/(M-m)
if Mc==mc:
  scalec=1.0
else:
  scalec=1.0/(Mc-mc)
if N==n:
  elacs=1.0
else:
  elacs=1.0/(N-n)
def pict(c):
  c.setFont('Helvetica',10*psc/500.0)
  drawnode=[]
  for a,b,e,d,f,g,i,j in coo:
    if g<eps:
      continue
    drawnode+=[i,j]
    str=unicode('%.2f'%g)
    c.setLineWidth(30*scale*(f-m)*psc/500.0)
    re,gr,bl=h2tuple(1-scalec*(g-mc))
    c.setStrokeColorRGB(re,gr,bl)
    c.line(psc*a+(psc/2.0),psc*b+(psc/2.0),psc*e+(psc/2.0),psc*d+(psc/2.0))
    c.translate((psc/2.0)*(a+e)+(psc/2.0),(psc/2.0)*(b+d)+(psc/2.0))
    c.rotate(abs(atan2(a-e,b-d))*180.0/pi-90)
    c.drawString(0,psc/100.0,str)
    c.rotate(-abs(atan2(a-e,b-d))*180.0/pi+90)
    c.translate(-(psc/2.0)*(a+e)-(psc/2.0),-(psc/2.0)*(b+d)-(psc/2.0))
  c.setStrokeColorRGB(1,1,1)
  c.setLineWidth(2*psc/500.0)
  for i in B.st:
    c.circle(psc*(i/ht)+(psc/2.0),psc*(i%ht)+(psc/2.0),psc/12.0,1,1)
  c.setFillGray(0.7)
  for i in B.nodes():
    if i in drawnode:
      c.circle(psc*(i/ht)+(psc/2.0),psc*(i%ht)+(psc/2.0),psc/25.0,1,1)
  for n in B.fr:
    if not [Fd[(n,s)] for s in ('x','y')]==[0.0,0.0]:
      arrow(c,n,Fd[(n,'x')],Fd[(n,'y')])
def arrow(c,x,xh,xv):
  c.setStrokeColor(black)
  c.setFillColor(black)
  h,v=elacs*(xh-n),elacs*(xv-n)
  c.setLineWidth(psc/100.0)
  sc=hypot(h,v)*psc/2.5
  i=x/ht
  j=x%ht
  c.translate(psc*i+(psc/2.0),psc*j+(psc/2.0))
  c.rotate(atan2(v,h)*180.0/pi)
  p=c.beginPath()
  p.moveTo(0,0)
  p.lineTo(sc,0)
```

```
      p.lineTo(sc-20*psc/500.0,20*psc/500.0)
      p.lineTo(sc-20*psc/500.0,-20*psc/500.0)
      p.lineTo(sc,0)
      c.drawPath(p,1,1)
      c.rotate(-atan2(v,h)*180.0/pi)
      c.translate(-psc*i-(psc/2.0),-psc*j-(psc/2.0))
    pict(c)
    c.showPage()
    c.save()
    print 'acroread %s'%pdf_fn
    print 'evince %s'%pdf_fn

def drawlast(filename='_truss_optimize_last.data'):
    ' Draws picture recovered from data file. '
    f=open(filename,'r')
    T=Truss()
    barlist,T.optd,T.Fd,T.L,T.lnth,T.hght,T.st,T.fr=cPickle.load(f)
    f.close()
    T.add_edges_from(barlist)
    barlist.sort()
    T.ls=barlist
    T.draw()

def coord(i,h):
    if (type(i),type(h))!=(int,int):
        print 'Taking integer part'
        i,h=int(i),int(h)
    return i/h,i%h

def drooc(x,y,h):
    if y>h:
        print 'y must be less than the height'
        return -1
    slash=[i/h for i in range((x+1)*h+y*h)]
    modulus=[i%h for i in range((x+1)*h+y*h)]
    re=[i for i in range((x+1)*h+y*h) if slash[i]==x and modulus[i]==y]
    if len(re)!=1:
        print 'Error'
    return re[0]

def example(no=0,fname='truss_example_'):
    numberofexamples=7
    if no>=numberofexamples:
        print 'List of examples available: ',range(numberofexamples)
    elif no==0:  ## 3X2 ##
        st=[0,4]
        y=1e6
        Fd={}
        Fd[(3,'y')]=-1000.0
        T=Truss(youngs_modulus=y,Fd=Fd,lnth=3,hght=2,dist=3,st=st)
        T.solve()
        T.draw(pdf_fn=fname+'%i.pdf'%no)
    elif no==1:  ## 2X5 ##
        st=[0,4]
        y=1000.0
        Fd={}
        Fd[(2,'x')]=-100.0
        T=Truss(youngs_modulus=y,Fd=Fd,lnth=2,hght=5,dist=3,st=st)
        T.solve()
        T.draw(pdf_fn=fname+'%i.pdf'%no)
    elif no==2:  ## 5X2 ##
        st=[0,8]
```

```
    y=1000.0
    Fd={}
    Fd[(3,'y')]=  50.0
    Fd[(3,'x')]=-100.0
    Fd[(7,'y')]=  50.0
    Fd[(7,'x')]= 100.0
    T=Truss(youngs_modulus=y,Fd=Fd,lnth=5,hght=2,dist=3,st=st)
    T.solve()
    T.draw(pdf_fn=fname+'%i.pdf'%no)
  elif no==3:  ## 3X5 ##
    st=[0,10]
    y=1000.0
    Fd={}
    Fd[(4,'y')]=-1.0
    Fd[(4,'x')]= 1.3
    T=Truss(youngs_modulus=y,Fd=Fd,lnth=3,hght=5,dist=2,st=st)
    T.solve()
    T.draw(pdf_fn=fname+'%i.pdf'%no)
  elif no==4:  ## 9X4 ##
    st=[0,32]
    Fd={}
    for i in range(4,32,4): Fd[(i,'y')]=-1000.0
    T=Truss(Fd=Fd,lnth=9,hght=4,dist=3,st=st)
    T.solve()
    T.draw(pdf_fn=fname+'%i.pdf'%no)
  elif no==5:  ## 15X6 ##
    st=[12,18,66,72]
    Fd={}
    for i in range(90)[2::6]: Fd[(i,'y')]=-200.0
    T=Truss(Fd=Fd,lnth=15,hght=6,dist=4,st=st)
    T.solve()
    T.draw(pdf_fn=fname+'%i.pdf'%no)
  elif no==6:  ## 30X10 ##
    print 'This may take a few minutes...'
    st=[20,30,260,270]
    Fd={}
    for i in range(300)[::10]:
      if i not in st: Fd[(i,'y')]=-1000.0
    T=Truss(Fd=Fd,lnth=30,hght=10,dist=4,st=st)
    T.solve()
    T.draw(pdf_fn=fname+'%i.pdf'%no)

if __name__=='__main__':
  if len(argv)==1:
    st=[(0,0),(2,0)]
    Fd={}
    Fd[((1,1),'y')]=-100
    B=Easytruss(Fd=Fd,lnth=3,hght=2,st=st)
    B.solve()
    B.draw()
  else:
    job=int(argv[1])
    example(job)
```

The Lovász number index of small graphs

[.pdf pictures provided by Dr. Keith M. Briggs.*]*

Figure 4.16: The ϑ number of all small graphs of up to 7 nodes.

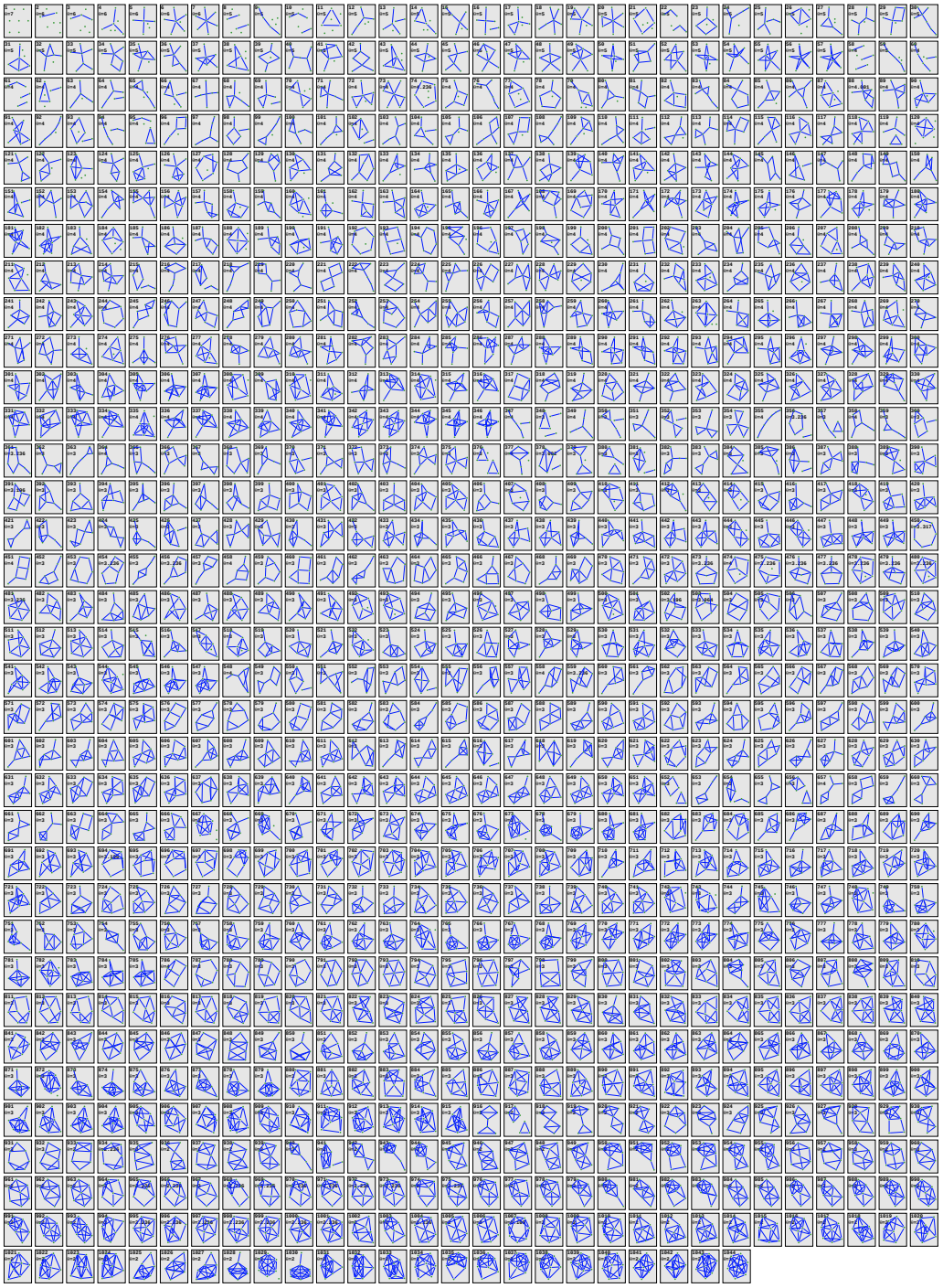

Figure 4.17: The ϑ number of all graphs of 7 nodes.

Bibliography

Steven J. Benson and Yinyu Ye. Algorithm 875:DSDP5: Software for semidefinite programming. *ACM Transactions on Mathematical Software*, 34(3), 2008. URL http://doi.acm.org/10.1145/1356052.1356057.

Steven J. Benson, Yinyu Ye, and Xiong Zhang. Solving large-scale sparse semidefinite programs for combinatorial optimization. *SIAM Journal on Optimization*, 10(2):443–461, 2000.

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

Robert M. Freund. Truss design and convex optimization. Technical report, Massachusetts Institute of Technology, April 2004. URL http://ocw.mit.edu/NR/rdonlyres/Sloan-School-of-Management/15-094JSpring-2004/2172F2C6-E4DB-45EB-AFD3-36C92E2AABB5/0/truss_design_art.pdf.

Matteo Frigo and Steven G. Johnson. The design and implementation of fftw3. Technical report, Massachusetts Institute of Technology, 2003. URL http://www.fftw.org/. http://www.fftw.org/fftw-paper-ieee.pdf.

M. Galassi, Jim Davies, and Brian Gough. Gnu scientific library reference manual (2nd ed.). Technical Report ISBN 0954161734, 2003. URL http://www.gnu.org/software/gsl/.

Walter W. Garvin. *Introduction to Linear Programming*. McGraw-Hill Book Company, Inc, 1960.

Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, 1985.

Donald E. Knuth. The sandwich theorem. *The Electronic Journal of Combinatorics*, 1(A1), 1994. URL http://www.emis.de/journals/EJC/Volume_1/volume1.html#A1.

L. Lovász. On the shannon capacity of a graph. *Information Theory, IEEE Transactions on*, 25(1):1–7, Jan 1979. ISSN 0018-9448.

Andrew Makhorin. Technical report, Department for Applied Informatics, Moscow Aviation Institute, Moscow, Russia, 2003. URL http://www.gnu.org/software/glpk/. The GLPK package is a part of the GNU project, released under the aegis of GNU. (C) Copyright 2000, 2001, 2002, 2003.

Lieven Vandenberghe, Stephen Boyd, and Shao-Po Wu. Determinant maximization with linear matrix inequality constraints. *SIAM Journal on Matrix Analysis and Applications*, 19(2):499–533, 1998. URL citeseer.ist.psu.edu/vandenberghe98determinant.html.

R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2), 2005. URL http://math-atlas.sourceforge.net/.

Wikipedia. King post — wikipedia, the free encyclopedia, 2008a. URL http://en.wikipedia.org/w/index.php?title=King_post&oldid=227514338. [Online; accessed 30-July-2008].

Wikipedia. Tyne bridge — wikipedia, the free encyclopedia, 2008b. URL http://en.wikipedia.org/w/index.php?title=Tyne_Bridge&oldid=229811490. [Online; accessed 21-August-2008].

Wikipedia. Young's modulus — wikipedia, the free encyclopedia, 2008c. URL http://en.wikipedia.org/w/index.php?title=Young%27s_modulus&oldid=232681599. [Online; accessed 21-August-2008].