Grado en Matemáticas

TRABAJO FIN DE GRADO

Introducción a la Programación Convexa Discplinada (DCP). Algunas aplicaciones y su resolución informática.

María Cabeza Cobano

Sevilla, Junio de 2019

Índice general

Resumen

En este trabajo de fin de grado se explica la Programación Convexa Disciplinada, una metodología para trabajar con problemas de optimización convexa. Su objetivo es simplificar el proceso de verificación de la convexidad de un problema, que en muchos casos es un trabajo intratable. Se basa en una librería de funciones, a partir de las cuales se construyen los problemas y un conjunto de normas impuesto sobre estas funciones, que establecen como combinarlas. No se restringe a un tipo determinado de forma estándar como hacen otras metodologías y no pierde generalidad ya que la librería en la que se basa es extensible.

Esta metodología se puede llevar a la práctica a través del paquete CVXR de R. Permite formular los problemas de una forma muy natural e intuitiva. En CVXR es posible formular y resolver problemas para los que no existe un código personalizado. Dado un problema, CVXR emplea la programación convexa disciplinada previamente a resolver, para probar que sea convexo y por tanto garantizar que la solución óptima es global.

Abstract

Convex Disciplined Programming is a methodology for working with convex optimization problems. The objective is to simplify the convexity verification, because this is often an intractable task. It is based on a library of functions, that we use for construct the problems, and a set of rules that constitute a set of sufficient conditions for guarantee convexity. DCP is not restricted to a standard way as other methodologies and it does not lose generality since the library is extensible.

This methodology can be put into practice through the package CVXR of R. It allows the user to formulate convex optimization problems in a natural and intuitive mathematical syntax. In CVXR it is possible to formulate and solve problems for which there is no personalized code. Given a problem, CVXR applies Disciplined Convex Programming to verify the problem's convexity and therefore guarantee that the optimal solution is global.

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Programación Matemática

La Programación Matemática es una parte de la Matemática Aplicada que trata de resolver problemas de decisión en los que se deben determinar acciones que optimicen el objetivo planteado, pero cumpliendo ciertas limitaciones o condiciones. Forma parte del campo de la Investigación Operativa.
El objetivo de la Programación Matemática es el estudio teórico de problemas de optimización, así como el establecimiento de los algoritmos de resolución.

Al igual que muchas otras ramas de las matemáticas, nace a partir de sus aplicaciones. Los orígenes de la Investigación Operativa son muy antiguos, ya que todas las sociedades siempre se han planteado el problema de optimizar sus recursos. Sin embargo, el concepto formal de Investigación Operativa no aparece formalmente hasta 1938 con Patrick Maynard Stuart Blackett, en el marco de investigaciones militares motivadas por la Segunda Guerra Mundial.
Fue en la década de 1940 cuando comienza a desarrollarse la Programación Matemática debido a que en estos años la competencia económica en la industria requería técnicas de optimización cada vez más avanzadas. George Bernard Dantzig y Leonid Vitálievich Kantoróvich son considerados los pioneros.
En las primeras décadas se desarrollaron una gran variedad de teoremas y teorías, y además, condujo a la revalorización de algunos resultados matemáticos antiguos como el lema de Farkas.

El método del simplex, desarrollado por G. B. Dantzig en 1947, hizo posible resolver varios tipos de problemas de optimización a gran escala, como problemas de transporte, producción o asignación de recursos.
Por otro lado, la interacción con las nuevas tecnologías de computación permitió la aplicación del método simplex a problemas mucho mayores.
Los problemas prácticos casi siempre involucran grandes cantidades de datos y cálculos que solo pueden ser abordados de forma razonable mediante la potencia de cálculo de un ordenador. Por lo que el desarrollo de la informática en los últimos años es el mayor responsable del auge de la Programación Matemática. Cada vez se desarrollan programas (solvers) más eficientes que permiten trabajar con problemas cada vez mayores. Este éxito conllevó a que se extendiera a otros campos tales como la industria, física, informática, economía, estadística y probabilidad, ecología, educación,. . . , siendo hoy en día utilizada

prácticamente en todas las áreas.

Un problema de programación matemática, en general, se formula como,

$$\begin{array}{lrl} \text{Minimizar} & f(x), & \\ \text{Sujeto a:} & g_i(x) & \leq 0,\ i = 1, ..., n, \\ & h_j(x) & = 0,\ j = 1, ..., m \\ & x & \in X \in \mathbb{R}^k \end{array} \tag{1.1}$$

donde x son las variables del problema; f, g y h son funciones reales de variable real. A f se le denomina función objetivo y $g_i(x) \leq 0, h_j(x) = 0$ son las restricciones de desigualdad e igualdad, respectivamente.

Se dice que:

- Un punto es factible si está en X y satisface las restricciones.
- Un punto es solución del problema si es factible y no existe otro punto factible donde el valor de la función objetivo sea menor.

Así se dirá que x^* es el óptimo si es factible y $f(x^*) \leq f(x)$ para cada x factible.

1.1.1. Resolver el problema

Un método de resolución para una clase de problemas de optimización es un algoritmo que calcula una solución dado un problema particular de la clase. Desde finales de la década de 1940, se ha dedicado un gran esfuerzo al desarrollo de algoritmos para resolver diversas clases de problemas de optimización, analizar sus propiedades y desarrollar buenas implementaciones de software. La efectividad de estos algoritmos varía considerablemente y depende de factores como las formas particulares de las funciones objetivo y de restricción, así como del número de variables y restricciones que existen. Incluso cuando las funciones objetivo y de restricción son dos veces diferenciables (por ejemplo, polinomios), el problema general de optimización es sorprendentemente difícil de resolver. Los enfoques para el problema general, por lo tanto, implican algún tipo de compromiso, como un tiempo de cómputo muy largo, o la posibilidad de no encontrar la solución. Algunos de estos métodos los veremos más adelante. Sin embargo, existen excepciones, para algunas clases de problemas hay algoritmos efectivos que pueden resolver de manera confiable incluso problemas con cientos de variables y restricciones. Dos ejemplos importantes y bien conocidos, son los problemas de mínimos cuadrados y los problemas de programación lineal.
La optimización convexa es otra excepción a la regla: al igual que los mínimos cuadrados o la programación lineal, existen algoritmos muy efectivos que pueden resolver de manera confiable y eficiente incluso problemas convexos grandes.

1.2. Programación convexa

La Programación Matemática se divide en varias ramas según las características de las variables y de las funciones que describan el objetivo y las restricciones de cada problema. Hablaremos de Programación Lineal cuando estas funciones son lineales en sus variables, en otro caso hablaremos de Programación No Lineal. Dentro de la Programación No Lineal

existen varios tipos de problemas, nos vamos a centrar especialmente en la Programación Convexa.

En un problema convexo tanto la función objetivo como las funciones que describen las restricciones de desigualdad son convexas, mientras que las funciones que describen las restricciones de igualdad son afines.

Como veremos en el capítulo 2, la Programación Convexa tiene numerosas aplicaciones y, debido a sus propiedades teóricas, posee significantes ventajas respecto a problemas no convexos, lo que le da una gran importancia. La ventaja más significativa es que en un problema convexo se puede garantizar que cualquier óptimo local es global. Esto, en general, no es posible y el trabajo de encontrar el óptimo global no es nada fácil en ese caso.
Pero por otra parte, reconocer los problemas de optimización convexos, o aquellos que pueden transformarse en problemas de optimización convexos puede ser un desafío. Para ciertos problemas concretos, cuya teoría está muy desarrollada, los problemas pueden ser identificados y resueltos con un esfuerzo relativamente pequeño. Sin embargo, este no es el caso de la programación convexa en general. Se requiere una comprensión profunda del análisis convexo. Además, se debe encontrar una manera de transformar el problema en una de las muchas formas estándar limitadas que veremos; o, en su defecto, desarrollar un solver personalizado. Estos requisitos constituyen una barrera considerable para el uso de la programación convexa.

El propósito de este trabajo es presentar una metodología que haga más sencillo el trabajo con problemas convexos. Introducimos en el capítulo 3 la Programación Convexa Disciplinada (DCP, Disciplined Convex Programming). Como sugiere el término "disciplinado", esta metodología impone un conjunto de convenciones que se deben seguir al construir programas convexos. Las convenciones son simples y fáciles de manejar, se basan en los principios básicos del análisis convexo. Los problemas que siguen estas convenciones se denominan programas convexos disciplinados o DCPs. Estas convenciones permiten que gran parte de las manipulaciones y transformaciones necesarias para analizar y resolver programas convexos sean automatizadas. Además, no limitan la generalidad de la programación convexa. Por ejemplo, la tarea de determinar si un programa no lineal arbitrario es convexo es teórica y prácticamente intratable, sin embargo, la tarea de determinar si es un programa convexo disciplinado es sencilla. Además, las transformaciones necesarias para convertir programas convexos disciplinados en una forma solucionable, así como las transformaciones inversas necesarias para recuperar información valiosa dual, pueden ser completamente automatizadas.

Introducimos también en este trabajo, en el capítulo 4, una descripción del paquete CVXR del entorno de trabajo R. Proporciona un lenguaje de modelado orientado a la resolución de problemas de programación convexa, que permite formular y resolver nuevos problemas para los cuales no existe un código personalizado. Posee una sintaxis simple y matemáticamente intuitiva, y es muy flexible, con pocas líneas de código se puede transformar un problema y volverlo a resolver rápidamente. CVXR aplica programación convexa disciplinada (DCP) para verificar la convexidad del problema.

Capítulo 2

Análisis convexo

En este capítulo vamos a exponer definiciones básicas y resultados del análisis convexo y la optimización convexa. La intención es cubrir los conceptos necesarios para el desarrollo de la optimización convexa disciplinada que veremos en el capítulo 3.
Damos una serie de reglas de cálculo convexo, es decir, operaciones entre conjuntos y entre funciones que preservan la convexidad. Combinar los ejemplos básicos con las reglas de cálculo convexo nos permite formar y reconocer algunos conjuntos y funciones convexas bastante complejas.

2.1. Conjuntos convexos

Definición

Un conjunto C es convexo si dados dos puntos $x_1, x_2 \in C$ cualesquiera, el segmento que une estos puntos está contenido en C, es decir, para cualquier θ tal que $0 \leq \theta \leq 1$ se tiene:

$$\theta x_1 + (1 - \theta)x_2 \in C \tag{2.1}$$

Cono convexo

Un conjunto C es un cono si para todo $x \in C$ se tiene que $\alpha x \in C$ para todo $\alpha > 0$. Si C es convexo, hablaremos de cono convexo.

2.1.1. Operaciones que preservan la convexidad

En esta sección describiremos algunas operaciones que preservan la convexidad de los conjuntos o que nos permiten construir nuevos conjuntos convexos.

Intersección

La convexidad se preserva bajo intersección: si S_1 y S_2 son convexos entonces $S_1 \cap S_2$ es convexo.

Demostración

Para dos puntos cualesquiera $x, y \in S_1 \cap S_2$ se tiene que:

- $x, y \in S_1 \Rightarrow \forall \alpha \in [0, 1]$, $\alpha x + (1 - \alpha)y \in S_1$ ya que S_1 es convexo.

- $x, y \in S_2 \Rightarrow \forall \alpha \in [0,1]$, $\alpha x + (1-\alpha)y \in S_2$ ya que S_2 es convexo.

Por tanto $\alpha x + (1-\alpha)y \in S_1 \cap S_2$, luego $S_1 \cap S_2$ es convexo.

Esta propiedad se extiende a la intersección de un número infinito de conjuntos: si S_α es convexo para cada $\alpha \in A$ entonces $\cap_{\alpha \in A} S_\alpha$ es convexo.

Suma

La suma de dos conjuntos se define como:

$$S_1 + S_2 = \{x_1 + x_2 \mid x_1 \in S_1, x_2 \in S_2\}$$

Si S_1 y S_2 son convexos entonces $S_1 + S_2$ es convexo.

Demostración

Para dos puntos cualesquiera $x, y \in S_1 + S_2$ se tiene que:

- $x \in S_1 + S_2 \Rightarrow x = x_1 + x_2$ con $x_1 \in S_1$, $x_2 \in S_2$.
- $y \in S_1 + S_2 \Rightarrow y = y_1 + y_2$ con $y_1 \in S_1$, $y_2 \in S_2$.

Sea $\alpha \in [0,1]$,

$$\begin{aligned}\alpha x + (1-\alpha)y =& \alpha x_1 + \alpha x_2 + (1-\alpha)y_1 + (1-\alpha)y_2 = \\ & \alpha x_1 + (1-\alpha)y_1 + \alpha x_2 + (1-\alpha)y_2 \in S_1 + S_2\end{aligned}$$

ya que $\alpha x_1 + (1-\alpha)y_1 \in S_1$ porque S_1 es convexo y $\alpha x_2 + (1-\alpha)y_2 \in S_2$ porque S_2 es convexo, queda demostrado que $S_1 + S_2$ es convexo.

Producto cartesiano

El producto cartesiano de dos conjuntos se define como:

$$S_1 \times S_2 = \{(x_1, x_2) \mid x_1 \in S_1, x_2 \in S_2\} \tag{2.2}$$

Si S_1 y S_2 son convexos entonces $S_1 \times S_2$ es convexo.

Demostración

Para dos puntos cualesquiera $x, y \in S_1 \times S_2$ se tiene que:

- $x \in S_1 \times S_2 \Rightarrow x = (x_1, x_2)$ con $x_1 \in S_1$, $x_2 \in S_2$.
- $y \in S_1 \times S_2 \Rightarrow y = (y_1, y_2)$ con $y_1 \in S_1$, $y_2 \in S_2$.

Sea $\alpha \in [0,1]$,

$$\begin{aligned}\alpha x + (1-\alpha)y &= \alpha(x_1, x_2) + (1-\alpha)(y_1, y_2) = \\ &= (\alpha x_1 + (1-\alpha)y_1, \alpha x_2 + (1-\alpha)y_2) \in S_1 \times S_2\end{aligned}$$

ya que $\alpha x_1 + (1-\alpha)y_1 \in S_1$ porque S_1 es convexo y $\alpha x_2 + (1-\alpha)y_2 \in S_2$ porque S_2 es convexo, queda demostrado que $S_1 \times S_2$ es convexo.

2.2. Funciones convexas

2.2.1. Definiciones

- Una función $f : \mathbb{R}^n \to \mathbb{R}$ es convexa si dom f es un conjunto convexo y para todo $x, y \in$ dom f, y para todo $\theta \in [0, 1]$ se tiene que:

$$f(\theta x + (1 - \theta)y) \leq \theta f(x) + (1 - \theta)f(y) \tag{2.3}$$

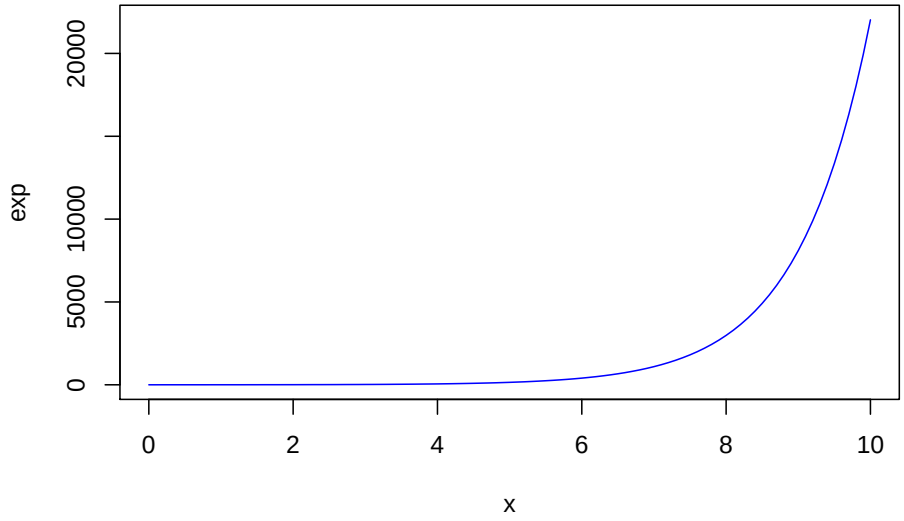

Figura 2.1: Función convexa.

- Una función $f : \mathbb{R}^n \to \mathbb{R}$ es cóncava si dom f es un conjunto convexo y para todo $x, y \in$ dom f, y para todo $\theta \in [0, 1]$ se tiene que:

$$f(\theta x + (1 - \theta)y) \geq \theta f(x) + (1 - \theta)f(y) \tag{2.4}$$

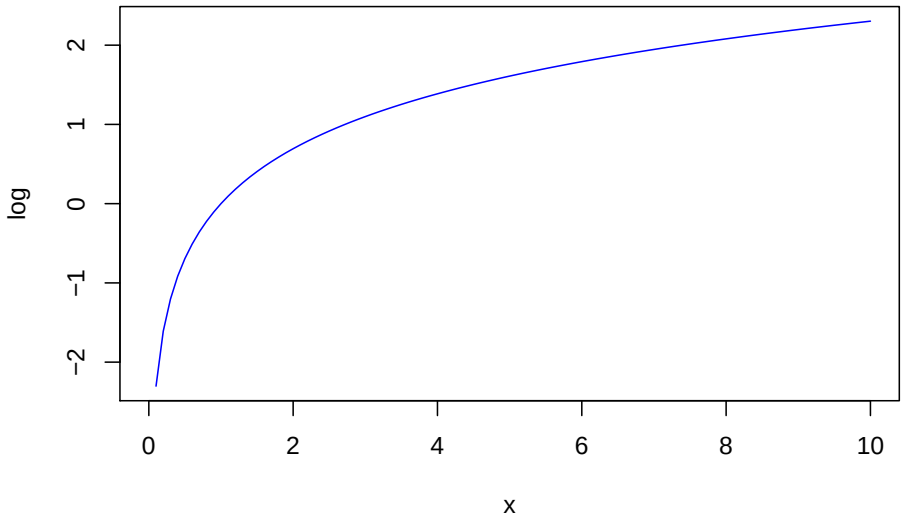

Figura 2.2: Función cóncava.

Se cumple que si f es cóncava entonces -f es convexa.

- Las funciones afines son aquellas para las que se da la igualdad, así que son tanto convexas como cóncavas. Es decir, una función $f : \mathbb{R}^n \to \mathbb{R}$ es afín si para todo $x, y \in$ dom f, y para todo $\theta \in [0, 1]$ se tiene que:

$$f(\theta x + (1 - \theta)y) = \theta f(x) + (1 - \theta)f(y) \tag{2.5}$$

Grafo

El grafo de una función $f : R^n \to \mathbb{R}$ es un subconjunto de $\mathbb{R}^{n+1}$ que se define como:

$$\{(x, f(x)) \mid x \in domf\} \tag{2.6}$$

Epigrafo

El epigrafo de una función $f : \mathbb{R}^n \to \mathbb{R}$ se define como:

$$Epi(f) = \{(x, t) \mid x \in domf, f(x) \leq t\} \tag{2.7}$$

Es el conjunto de puntos situado sobre la curva.

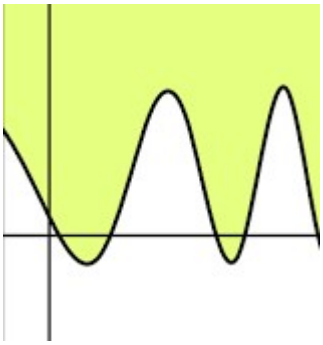

Figura 2.3: Epigrafo

Hipografo

El hipografo de una función $f : \mathbb{R}^n \to \mathbb{R}$ se define como:

$$Hip(f) = \{(x,t) \mid x \in domf, t \leq f(x)\} \tag{2.8}$$

Es el conjunto de puntos situado bajo la curva.

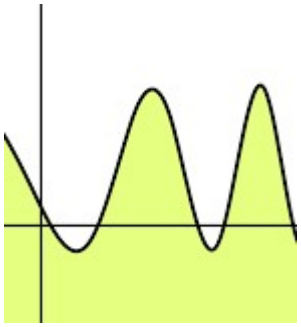

Figura 2.4: Hipografo

2.2.2. Propiedades básicas

Un principio fundamental del análisis convexo es la relación entre funciones convexas y conjuntos convexos, y se establece a través del epigrafo e hipografo de las funciones.

Lema. Una función $f : \mathbb{R}^n \to \mathbb{R}$ es convexa si y solo si su epigrafo es un conjunto convexo.

Demostración

Supongamos que f es convexa, entonces sean $(x,t),(y,s), \in Epi(f)$ se tiene que para todo $\lambda \in [0,1]$:

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) \leq \lambda t + (1-\lambda)s$$

Por tanto, $\lambda(x,t) + (1-\lambda)(y,s) \in Epi(f)$, por tanto se tiene que Epi(f) es convexo.

Supongamos ahora que Epi(f) es un conjunto convexo. Dados $x, y \in dom\ f$ por definición de Epi(f) se tiene que $(x, f(x)), (y, f(y)) \in Epi(f)$.

Como Epi(f) es convexo, $\lambda(x, f(x)) + (1-\lambda)(y, f(y)) \in Epi(f)$. Desarrollando esta expresion obtenemos:

$$\lambda(x, f(x)) + (1-\lambda)(y, f(y)) = (\lambda x + (1-\lambda)y, \lambda f(x) + (1-\lambda)f(y)) \in Epi(f)$$

Por tanto,

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$$

es decir, f es convexa.

Lema. Una función $f : \mathbb{R}^n \to \mathbb{R}$ es cóncava si y solo si su hipografo es un conjunto convexo.

Demostración

Supongamos que f es cóncava, entonces sean $(x,t), (y,s) \in Hip(f)$ se tiene que para todo $\lambda \in [0,1]$:

$$-f(\lambda x + (1-\lambda)y) \leq \lambda(-1)f(x) + (1-\lambda)(-1)f(y) \Rightarrow$$
$$f(\lambda x + (1-\lambda)y) \geq \lambda f(x) + (1-\lambda)f(y) \geq \lambda t + (1-\lambda)s$$

Por tanto, $\lambda(x,t) + (1-\lambda)(y,s) \in Hip(f)$, por tanto se tiene que Hip(f) es convexo.

Supongamos ahora que Hip(f) es un conjunto convexo. Dados $x, y \in dom\ f$ por definición de Hip(f) se tiene que $(x, f(x)), (y, f(y)) \in Hip(f)$.
Como Hip(f) es convexo, $\lambda(x, f(x)) + (1-\lambda)(y, f(y)) \in Hip(f)$. Desarrollando esta expresión obtenemos:

$$\lambda(x, f(x)) + (1-\lambda)(y, f(y)) = (\lambda x + (1-\lambda)y, \lambda f(x) + (1-\lambda)f(y)) \in Hip(f)$$

Por tanto,

$$f(\lambda x + (1-\lambda)y) \geq \lambda f(x) + (1-\lambda)f(y)$$

es decir, f es cóncava.

Estas relaciones se pueden expresar también de la siguiente forma, suponiendo f una función convexa y g cóncava:

$$f(x) = inf\{y \mid (x,y) \in Epi(f)\}$$
$$g(x) = sup\{y \mid (x,y) \in Hip(g)\}$$

2.2.3. Operaciones que preservan la convexidad

En esta sección describiremos algunas operaciones que preservan la convexidad o concavidad de las funciones o que nos permiten construir nuevas funciones convexas o cóncavas.

Suma

Si f_1 y f_2 son funciones convexas, entonces también lo es su suma $f_1 + f_2$.

Demostración

Como f_1 y f_2 son convexas, dados $x, y \in \text{dom } (f_1 + f_2)$ se tiene que para todo $\lambda \in [0, 1]$:

$$f_1(\lambda x + (1-\lambda)y) \leq \lambda f_1(x) + (1-\lambda) f_1(y)$$
$$f_2(\lambda x + (1-\lambda)y) \leq \lambda f_2(x) + (1-\lambda) f_2(y)$$

Entonces,

$$\begin{aligned}(f_1 + f_2)(\lambda x + (1-\lambda)y) &= f_1(\lambda x + (1-\lambda)y) + f_2(\lambda x + (1-\lambda)y) \leq \\ &\leq \lambda f_1(x) + (1-\lambda) f_1(y) + \lambda f_2(x) + (1-\lambda) f_2(y) = \\ &= \lambda(f_1(x) + f_2(x)) + (1-\lambda)(f_1(y) + f_2(y))\end{aligned}$$

Luego, $f_1 + f_2$ es convexa.

Producto por escalar

Si f es una función convexa y $\alpha \geq 0$, entonces αf es convexa.

Demostración

Dados $x, y \in$ dom f se tiene que para todo $\lambda \in [0, 1]$:

$$\begin{aligned}f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y) \Rightarrow \\ \alpha f(\lambda x + (1-\lambda)y) \leq \alpha(\lambda f(x) + (1-\lambda) f(y)) = \lambda(\alpha f(x)) + (1-\lambda)(\alpha f(y))\end{aligned}$$

Luego, αf es convexa.

Transformaciones afines

Suponiendo $f : \mathbb{R}^n \to \mathbb{R}, A \in \mathbb{R}^{n \times m}$ y $b \in \mathbb{R}$. Se define $g : \mathbb{R}^m \to \mathbb{R}$ como

$$g(x) = f(Ax + b) \tag{2.9}$$

con dom g $=\{\ x \mid Ax + b \in dom\ f\}$. Entonces si f es convexa también lo es g. De la misma forma, cuando f es cóncava también lo es g.

Demostración

Dados $x, y \in$ dom g cualesquiera, entonces para todo $\lambda \in [0, 1]$ se tiene:

$$\begin{aligned}&g(\lambda x + (1-\lambda)y) = f(A(\lambda x + (1-\lambda)y) + b) = && \text{(Por definición de g)} \\ &= f(\lambda(Ax + b) + (1-\lambda)(Ay + b)) \leq && \text{(Descomponiendo } b = \lambda b + (1-\lambda)b) \\ &\leq \lambda f(Ax + b) + (1-\lambda) f(Ay + b) = && \text{(f es convexa)} \\ &= \lambda g(x) + (1-\lambda) g(y)\end{aligned}$$

Composición

Vamos a estudiar las condiciones de $f : \mathbb{R} \to (\mathbb{R} \cup +\infty)$ y $g : \mathbb{R}^n \to (\mathbb{R} \cup +\infty)$ que garantizan la convexidad o concavidad de $h = f \circ g$.

Lema : Si $f : \mathbb{R} \to (\mathbb{R} \cup +\infty)$ es convexa y no decreciente y $g : \mathbb{R}^n \to (\mathbb{R} \cup +\infty)$ es convexa, entonces $h = f \circ g$ es convexa.

Demostración

Dados $x, y \in$ dom g se tiene que para todo $\lambda \in [0,1]$:

$$\begin{aligned}
&(f \circ g)(\lambda x + (1-\lambda)y) = f(g(\lambda x + (1-\lambda)y)) \leq \\
&\leq f(\lambda g(x) + (1-\lambda)g(y)) && \text{(g convexa y f no decreciente)} \\
&\leq \lambda f(g(x)) + (1-\lambda)f(g(y)) = && \text{(f convexa)} \\
&= \lambda(f \circ g)(x) + (1-\lambda)(f \circ g)(y)
\end{aligned}$$

Luego, $f \circ g$ es convexa.

Se obtienen resultados similares para funciones cóncavas y/o no crecientes:

- Si $f : \mathbb{R} \to (\mathbb{R} \cup +\infty)$ es convexa y no creciente y $g : \mathbb{R}^n \to (\mathbb{R} \cup -\infty)$ es cóncava, entonces $h = f \circ g$ es convexa.

- Si $f : \mathbb{R} \to (\mathbb{R} \cup -\infty)$ es cóncava y no decreciente y $g : \mathbb{R}^n \to (\mathbb{R} \cup -\infty)$ es cóncava, entonces $h = f \circ g$ es cóncava.

- Si $f : \mathbb{R} \to (\mathbb{R} \cup -\infty)$ es cóncava y no creciente y $g : \mathbb{R}^n \to (\mathbb{R} \cup +\infty)$ es convexa, entonces $h = f \circ g$ es cóncava.

Además se tienen reglas similares para las funciones con múltiples argumentos.

2.3. Problemas de optimización convexa

Un problema de optimización convexa es de la forma:

$$\begin{array}{rc}
\text{Minimizar} & f_0(x) \\
\text{Sujeto a:} & f_i(x) \leq 0, \quad i = 1, ..., n \\
& a_j^T x = b_j, \quad j = 1, ..., m \\
& x \in X
\end{array} \tag{2.10}$$

El vector x son las variables del problema, contenida en un conjunto $X \subseteq \mathbb{R}^k$. Tanto la función objetivo $f_0 : \mathbb{R}^k \to \mathbb{R}$, como las funciones que describen las restricciones $f_1, ..., f_n : \mathbb{R}^k \to \mathbb{R}$ son convexas.

Se dice que un punto es factible para el problema si satisface todas las restricciones. El problema es factible si existe al menos un punto factible e infactible en caso contrario.

El valor óptimo para el problema que hemos definido viene dado por:

$$\begin{aligned}
p^* = inf\{f_0(x) \mid f_i(x) \leq 0, i = 1, ..., m \\
a_i^T x = b_i, i = 1, ..., n\}
\end{aligned} \tag{2.11}$$

Si el problema es infactible consideramos $p^* = \infty$, ya que el ínfimo de un conjunto vacío es ∞.
En el caso en que haya puntos factibles x_k tal que $f_0(x_k) \to -\infty$ cuando $k \to \infty$ entonces decimos que el problema es ilimitado.

Decimos que x^* es la solución óptima si x^* es factible y $f(x^*) = p^*$.

El conjunto de problemas convexos (CP, convex problem) es un subconjunto estricto de los problemas de programación no lineal (NLP, nonlinear programming). Un problema no

lineal es un problema de optimización donde la función objetivo o alguna de las funciones de las restricciones no son lineales.

Dentro de la definición general del problema de optimización convexa se encuentran casos especiales como el problema de mínimos cuadrados (LS, least squares), los problemas de programación lineal (LP, linear programming), los problemas de programación semidefinida (SDP, semidefinite program) [71], los problemas cuadráticos (QP, quadratic program), los problemas de cono de segundo orden (SOCP, second-order cone program) [45], y los problemas de programación geométrica (GP, geometric programming) ([23], [5]).

Vamos a definir cada uno de estos tipos de problemas explicitando su formulación.

2.3.1. Problema lineal

El problema lineal es el tipo de problema más popular y más estudiado. Se formula de la siguiente forma:

$$\begin{array}{ll} \text{Minimizar} & c^T x, \\ \text{Sujeto a:} & a_i^T x \ \leq b_i, \ i = 1, ..., n, \end{array} \tag{2.12}$$

donde $c, a_1, ..., a_n \in \mathbb{R}^k$, $b_1, ..., b_n \in \mathbb{R}$ son parámetros del problema.

2.3.2. Problema de mínimos cuadrados

El problema de mínimos cuadrados es un problema sin restricciones que se formula como:

$$\text{Minimizar} \quad \| \ Ax + b \ \|_2^2 \tag{2.13}$$

donde $A \in \mathbb{R}^{n \times k}$, $b \in \mathbb{R}^n$ son los parámetros del problema.

2.3.3. Problema cuadrático

Un problema de optimización es cuadrático si la función objetivo es cuadrática y las restricciones son afines.

$$\begin{array}{ll} \text{Minimizar} & c^T x + \frac{1}{2} x^T Q x \\ \text{Sujeto a:} & Ax = b \end{array} \tag{2.14}$$

donde $Q \in S_+^k$ (S_+^k es el conjunto de las matrices cuadradas de dimensión $k \times k$ definidas positivas), $c \in \mathbb{R}^k$,$A \in \mathbb{R}^{n \times k}$, $b \in \mathbb{R}^n$ son los parámetros del problema.

2.3.4. Problema de cono de segundo orden

El problema de cono de segundo orden se formula como:

$$\begin{array}{lll} \text{Minimizar} & f^T x & \\ \text{Sujeto a:} & \| \ A_i x + b_i \ \|_2 & \leq c_i^T x + d_i, \ i = 1, ..., m \\ & Fx = g & \end{array} \tag{2.15}$$

donde $A_1, ..., A_n \in \mathbb{R}^{n_i \times n}$, $F \in \mathbb{R}^{p \times n}$, $b \in \mathbb{R}^{n_i}$, $c_1, ..., c_m, d_1, ..., d_m \in \mathbb{R}^n$, $g \in \mathbb{R}^p$ son los parámetros del problema.

2.3.5. Problemas de programación semidefinida

El problema de programación semidefinida se formula como:

$$\begin{array}{lll} \text{Minimizar} & & c^T x \\ \text{Sujeto a:} & x_1 F_1 + ... + x_n F_n + G & \leq 0, \\ & Ax & = b \end{array} \tag{2.16}$$

donde $F_1, ..., F_n, G \in S_+^k$ (S_+^k es el conjunto de las matrices cuadradas de dimensión $k \times k$ definidas positivas), $A \in \mathbb{R}^{p \times k}$, $b \in \mathbb{R}^p$ son los parámetros del problema.

2.3.6. Problemas de programación geométrica

Un problema de programación geométrica se formula como:

$$\begin{array}{lll} \text{Minimizar} & f_0(x) & \\ \text{Sujeto a:} & f_i(x) & \leq 1, \ i = 1, ..., m \\ & h_i(x) & = 1, \ i = 1, ..., p \end{array} \tag{2.17}$$

donde $f_0, ..., f_m$ son de la forma $f_i(x_1, ..., x_n) = \sum_{k=1}^{L} c_k x_1^{a_1} ... x_n^{a_n}$ y $h_1, ..., h_p$ son funciones monomiales.

2.4. Algoritmos numéricos

Gran parte del estudio de algoritmos numéricos en las últimas décadas se ha centrado en los métodos de punto interior ([84], [68]). Inicialmente, dicho trabajo se limitó a LPs ([40], [60]), pero pronto se amplió para abarcar otros CPs ([51], [52]). Estos métodos consisten en la búsqueda del óptimo a través de caminos que recorren el interior de la región factible.
Existen varios solvers excelentes disponibles, tanto comerciales como de distribución gratuita, vamos a enumerar algunos de ellos. Se dividen en dos clases: los que se basan en las formas estándar y los que se basan en un código personalizado.

2.4.1. Formas estándar

La mayoría de los solvers para programación convexa están diseñados para una familia limitada de problemas con una estructura específica, o que obedecen a ciertas convenciones, se conocen como formas estándar. Dos ejemplos comunes pueden ser el problema de mínimos cuadrados o los problemas lineales. La mayoría de los problemas convexos se pueden transformar en una o más de estas formas estándar. Sin embargo, las transformaciones requeridas a menudo están lejos de ser obvias.

Programas convexos smooth

Se han diseñado varios solvers para resolver problemas convexos en forma estándar de problemas no lineales, bajo la condición de que la función objetivo y las funciones de

restricciones de desigualdad sean dos veces diferenciables al menos en la región en que el algoritmo desea buscar. A estos problemas los llamaremos CP smooth, mientras que a los CP que no lo cumplan los llamaremos CP nonsmooth.

Entre los paquetes de software que resuelven CP smooth encontramos LOQO ([73]) (emplea un método primal-dual), y el paquete comercial MOSEK ([50]) (implementa el algoritmo homogéneo).

Una dificultad práctica en el uso de solvers de CP smooth es que el solver debe poder calcular el gradiente y el hessiano del objetivo y las funciones de restricción de desigualdad en los puntos elegidos. En algunos casos, para realizar estos cálculos, puede ser necesario un código personalizado. Muchos marcos de modelado simplifican este proceso en la mayoría de los casos al permitir que las funciones se expresen en forma matemática natural y que calculen las derivadas automáticamente ([24], [13]).

Programas cónicos

Una familia de formas estándar que se ha vuelto muy común son las forma cónicas primales y duales:

$$\begin{array}{llll} \text{Minimizar} & c^T x & \text{Minimizar} & b^T y \\ \text{Sujeto a:} & Ax = b \quad o & \text{Sujeto a:} & A^T y + z = c \\ & x \in K & & z \in K^* \end{array} \tag{2.18}$$

donde los conjuntos K y K^* son conos convexos cerrados. La forma cónica más común es la LP donde K=K^* es:

$$K = K^* = \mathbb{R}^n_+ = \{x \in \mathbb{R}^n \mid x_i \geq 0,\ i = 1, ..., n\} \tag{2.19}$$

Se puede demostrar que prácticamente cualquier CP puede representarse en forma cónica, con la elección adecuada de K o K^* ([52]). En la práctica, dos formas cónicas (además del LP) dominan los recientes estudios e implementaciones. Uno es el programa semidefinido (SDP), para el cual K = K^* es un isomorfismo del cono de matrices semidefinidas positivas.

$$S^n_+ = \{X = X^T \in \mathbb{R}^{n \times n} \mid \lambda_{min}(X) \geq 0\} \tag{2.20}$$

Otro es el programa de cono de segundo orden, para el cual K=K^* es el producto cartesiano de uno o más conos de segundo orden (o de Lorentz).

$$K = Q^{n_1} \times \cdots \times Q^{n_k}, \quad Q^n = \{(x, y) \in \mathbb{R}^n \times \mathbb{R} \mid \ \| x \|_2 \leq y\} \tag{2.21}$$

Para SDP y SOCP se han descubierto muchas aplicaciones y su geometría admite ciertas optimizaciones algorítmicas útiles ([53], [31]). Los solvers disponibles públicamente para SDP y SOCP incluyen SeDuMi [65], CDSP [11], SDPA [27], SDPT3 [69] y DSDP [8]. Son generalmente bastante eficientes, confiables y están completamente basados en datos: es decir, no requieren código externo para realizar cálculos de funciones.

Programación geométrica

Un problema de programación geométrica es en realidad un caso un tanto único, ya que en realidad no es convexo, pero una transformación simple produce un problema

equivalente que es convexo. En forma convexa, las funciones de restricción de objetivo y desigualdad obtienen una estructura denominada "log-sum-exp"; por ejemplo:

$$f(x) = log \sum_{k=1}^{M} e^{a_k^T + b_k}, \quad a_k \in \mathbb{R}^n, \quad b_k \in \mathbb{R}, \quad k = 1, ..., M \tag{2.22}$$

En la forma convexa son smooth CPs. Avances recientes en algoritmos especializados han mejorado bastante la eficiencia de su solución. [41]

2.4.2. Código personalizado

Hay casos en los que un CP no se puede transformar en una de las formas estándar anteriores. Una alternativa es utilizar uno de los métodos que enumeramos aquí, que son universales en el sentido de que, en teoría, pueden aplicarse a cualquier CP. El costo de esta universalidad es que el usuario debe determinar ciertas construcciones matemáticas y escribir un código personalizado para implementarlas.

Método de barrera

Reemplaza el conjunto de restricciones de desigualdad

$$S = \{x \in \mathbb{R}^n \mid f_i(x) \leq 0, \ i = 1, ..., n_g\} \tag{2.23}$$

mediante una función barrera dos veces diferenciable que satisface que su dominio es el interior de s, produciendo un problema modificado.

$$\begin{array}{ll} \text{Minimiza} & f(x) + t^* f_i(x) \\ \text{sujeto a} & h_j(x) = 0 \quad j = 1, 2..., n_h \end{array} \tag{2.24}$$

Bajo ciertas condiciones, la solución a este problema modificado converge a la del problema original cuando $t \to 0$. Cada iteración de un método de barrera realiza de manera efectiva los pasos de minimización de Newton en el problema para valores decrecientes de t. Un desarrollo completo del método de barrera se da en [52].

Hay varios obstáculos prácticos para el uso de un método de barrera. Uno de los principales es determinar una función de barrera válida no siempre es trivial, particularmente si las restricciones de desigualdad no son diferenciables. Además, el usuario debería proporcionar el código para calcular el valor y las derivadas de la función de barrera.

Métodos de plano de corte

Los métodos de localización o plano de corte, como el ACCPM [57] no requieren información derivada de las funciones f y g_i, sino que se basa únicamente en los planos de corte para restringir el conjunto de búsqueda. El usuario debe proporcionar un código para calcular subgradientes o planos de corte. El rendimiento de estos métodos suele ser inferior al de los otros mencionados, pero son ideales para usar cuando no se puede calcular la segunda derivada o es difícil.

2.5. Ventajas de los problemas convexos

La ventaja más importante de los problemas convexos respecto al resto de problemas no lineales es que, para estos, se garantiza que los óptimos locales son globales mientras que para los NLP, en general no se puede garantizar. Es importante porque si el problema presenta múltiples óptimos locales, encontrar el óptimo global puede ser complejo.
Por otra parte, la programación convexa también tiene una teoría de dualidad muy rica. Sabemos que el dual de un CP es un CP, por lo que resolver el problema dual, a menudo, proporciona información útil sobre el problema original. Por ejemplo, si el problema dual es ilimitado, entonces el original es infactible. Bajo ciertas condiciones, la implicación inversa también es cierta: si un problema no es factible, entonces su dual es ilimitado. Estas y otras consecuencias de la dualidad facilitan la construcción de algoritmos numéricos para conocer si un problema es infactible o ilimitado. [62]
Otra propiedad importante de los CP es la existencia demostrable de algoritmos eficientes para resolverlos. Nesterov y Nemirovsky demostraron que se puede construir un método de barrera de tiempo polinomial para cualquier CP que cumpla ciertas condiciones técnicas [51]. Otros autores han demostrado que los problemas que no cumplen esas condiciones pueden integrarse en problemas más grandes que hacen que los métodos de barrera sean universales ([85]).
Tengamos en cuenta que las propiedades teóricas analizadas, incluida la existencia de métodos de solución eficientes, se mantienen incluso si un CP no es diferenciable, es decir, si una o más de las funciones de restricción u objetivo no son diferenciables.

2.6. Convexidad y diferenciabilidad

Muchos solvers para NLPs smooth no convexos se pueden usar para CPs smooth, por lo que a veces no es estrictamente necesario conocer si el problema es convexo. Pero cuando el problema no es dos veces diferenciable es muy distinto. Los mejores métodos para resolver NLPs nonsmooth son mucho menos precisos, confiables o eficientes.

En general, es mejor no trabajar directamente con problemas que no son dos veces diferenciables, en lugar de ello es recomendable eliminar los puntos de no-diferenciabilidad reemplazándolos con variables y expresiones booleanas. Pero no siempre es sencillo, e introduce complejidades prácticas significativas de un tipo diferente.
Por el contrario, no hay nada en teoría que impida que un CP nonsmooth se resuelva tan eficientemente como un CP smooth. Por ejemplo, la prueba proporcionada por Nesterov y Nemirovsky de la existencia de las funciones de barrera para los CP no dependen de que el problema sea smooth, y los CP nonsmooth a menudo se pueden convertir en un problema smooth equivalente con una transformación cuidadosamente elegida. Por supuesto, ni la construcción de una función de barrera válida ni la transformación de diferenciabilidad suelen ser obvias.

Los CPs nonsmooth son muy comunes en la práctica, por ejemplo, la mayoría de los SDP y SOCP no triviales, o funciones convexas comunes, como el valor absoluto y la mayoría de las normas. De hecho, ser smooth es la excepción no la norma. Por lo tanto, encontrar una metodología de programación convexa que resuelva problemas nonsmooth proporcionaría un gran beneficio práctico. Si se puede lograr tal solución, entonces la distinción a priori entre convexidad y no convexidad se vuelve mucho más importante, porque la necesidad de evitar la no diferenciabilidad permanece solo en el caso no convexo.

2.7. Verificación de convexidad

Dados los beneficios de conocer la convexidad, sería muy útil poder verificar de forma automática si un programa matemático es o no convexo. Desafortunadamente este trabajo es, en general, muy difícil. Cada enfoque práctico del problema conlleva comprometer la generalidad, confiabilidad o automatización:

- Generalidad: el enfoque admite solo un subconjunto bien definido de programas convexos.
- Confiabilidad: el enfoque no hace determinaciones concluyentes en algunos casos.
- Automatización: el enfoque requiere al menos algo de ayuda del usuario.

Vamos a presentar una serie de estos intentos prácticos.

El enfoque mas ambicioso computacionalmente quizás sea el desarrollado independientemente por Crusius ([17]) (refinado e integrado en un producto comercial) y Orban and Fourer ([54]). Aunque estos sistemas se desarrollaron de manera independiente, y cada uno incluye características únicas, son bastante similares en concepto. Proceden en dos etapas. En primer lugar, se analizan las restricciones para obtener límites de la región factible. En la segunda etapa, se analizan las subexpresiones clave de las funciones objetivo y de restricción para determinar si son convexas en esta región factible aproximada. Esto se logra al diferenciar simbólicamente las expresiones y delimitar los valores de sus hessianos. El análisis produce uno de estos tres resultados:

- todas los hessianos son semidefinidos positivos sobre la región factible: el problema es convexo.
- cualquier hessiano es definido negativo en la región factible: el problema no es convexo.
- en otro caso: no se puede hacer una determinación concluyente.

Tengamos en cuenta que el tercer caso no significa que el problema sea convexo o no convexo; solo que el sistema no pudo llegar a una conclusión.

Limitar el alcance a una o más formas estándar produce resultados más confiables. Por ejemplo, muchos marcos de modelado determinan automáticamente si un modelo es un LP, lo que permite seleccionar algoritmos especializados para ellos ([24], [13]). Las herramientas de modelado como SDPSOL y LMITOOL emplean enfoques similares para verificar automáticamente los SDP ([30], [81]). Estos enfoques son efectivos porque estas formas estándar particulares se pueden reconocer completamente a través de un análisis de su estructura, determinando si se ajusta a un conjunto de reglas establecidas. Son perfectamente confiables y realizan determinaciones concluyentes en todos los casos. Pero, por supuesto, la generalidad es significativamente comprometida. Además, estos sistemas no intentan reconocer aquellos problemas que se pueden transformar en la forma estándar admitida.

Otra alternativa es la proporcionada por el sistema MPROBE ([15]), que emplea muestreo numérico para determinar empíricamente las formas de las funciones de restricción y objetivo. En muchos casos desaprueba de manera concluyente la linealidad o la convexidad, pero nunca puede demostrar de manera concluyente la convexidad. Su autor

promueve a MPROBE como una herramienta útil para ayudar al usuario a tomar sus propias decisiones. Es una herramienta poderosa para ese propósito, y puede que este enfoque interactivo pueda proporcionar una determinación de la convexidad prácticamente suficiente en algunos casos.

Estos enfoques prácticos para la verificación automática de la convexidad comprometen la generalidad, ya sea debido a las limitaciones de los algoritmos o por restricciones de usabilidad. Como veremos a continuación, la programación convexa disciplinada hace concesiones de un tipo diferente, recuperando la generalidad.

2.8. Aplicaciones

Se conoce una amplia variedad de aplicaciones prácticas para la programación convexa y la lista sigue creciendo. La Teoría de Control es quizás el área donde la aplicación de la programación convexa esté más desarrollada. Otras de las más destacadas son:

- Robótica ([14])
- Análisis de patrones y minería de datos, incluyendo máquinas de vectores de soporte ([66], [86])
- Optimización combinatoria y teoría de grafos ([2], [47])
- Optimización estructural ([1], [6], [7])
- Geometría algebraica ([43], [42])
- Procesamiento de la señal ([39], [82])
- Comunicaciones y teoría de la información ([19], [59])
- Redes ([9], [10])
- Diseño de circuitos ([72], [38])
- Computación cuántica ([21])
- Redes neuronales ([56])
- Ingeniería Química ([61])
- Economía y Finanzas ([32])

Una fuente prometedora de nuevas aplicaciones para la programación convexa es la extensión y mejora de las aplicaciones existentes para la programación lineal. Un ejemplo de esto es la programación lineal robusta, que permite que las incertidumbres en los coeficientes de un modelo de LP se tengan en cuenta en la solución del problema, transformándolo en un CP no lineal ([45]). Este enfoque produce soluciones robustas más rápidamente, y posiblemente más confiables. Presumiblemente, la programación lineal robusta sería aplicable en cualquier lugar donde actualmente se emplee la programación lineal, y donde la incertidumbre en el modelo plantea una preocupación importante.

Podríamos pensar que nuestro pronóstico sobre la utilidad de la programación convexa es optimista, pero hay buenas razones para creer que, de hecho, se está subestimando el número de aplicaciones. Podemos apelar a la historia de la programación lineal como precedente. George Dantzig publicó por primera vez su invento del método símplex para programación lineal en 1947; y aunque pronto se encontraron varias aplicaciones militares, no fue hasta 1955-1960 que el campo experimentó un crecimiento robusto ([18]). Este retraso se debió en gran parte a la escasez de recursos computacionales adecuados; pero ese es el punto: el descubrimiento de nuevas aplicaciones se aceleró solo una vez que los

avances de hardware y software hicieron que fuera realmente práctico resolver los LP. Del mismo modo, entonces, hay buenas razones para creer que la cantidad de aplicaciones conocidas para la programación convexa aumentará drásticamente si se puede hacer mas fácil el hecho de crear, analizar y resolver CPs.

Capítulo 3

Programación Convexa Disciplinada

La Programación Convexa Disciplinada pretende desarrollar una estrategia que permita verificar si un problema dado es convexo de la forma más sencilla posible. A continuación vamos a exponer esta estrategia siguiendo la tesis escrita por Michael Charles Grant (2004) ([33]) y el posterior artículo Disciplined Convex Programming (2006) escrito por Michael Grant, Stephen Boyd, and Yinyu Ye ([35]).
Se basa en dos elementos, una librería de funciones y conjuntos cuyas propiedades de convexidad se conocen y un conjunto de reglas sobre estos elementos. En la librería se declara explícitamente su curvatura (convexa / cóncava / afín), monotonía, dominio y signo. El objetivo es combinarlos y manipularlos de manera que produzcan resultados convexos, para ello se establece un conjunto de reglas, extraídas de los principios básicos del análisis convexo, se vieron en el capítulo 2 las principales. Establecen cómo se pueden combinar las funciones, los conjuntos, las variables, los parámetros y los valores numéricos para producir resultados convexos. La librería es extensible, es decir, la función o conjunto creada mediante estas reglas a partir de las funciones y conjuntos básicos se puede agregar a la librería para ser reutilizada en otros modelos.
En este capítulo se expone este conjunto de reglas y se verá como se aplican para llevar a cabo la verificación del problema. La librería se verá en el capítulo 4 con más detalle.

Un programa convexo disciplinado válido es simplemente un programa matemático construido de acuerdo al conjunto de reglas de convexidad utilizando elementos de la librería.

El conjunto de reglas de convexidad, introducido a continuación, constituye un conjunto de condiciones suficientes para garantizar la convexidad, no necesarias. Es decir, para cualquier problema matemático construido siguiendo estas reglas podemos asegurar que es convexo. Sin embargo, no podemos decir que un programa no es convexo si no se adapta a ellas, simplemente significará que no es DCP válido. Si queremos utilizar esta metodología tendríamos que reescribirlos de acuerdo a las reglas.

El hecho de que tengamos un conjunto de reglas determinado y una librería de funciones y conjuntos fijos, no limita la generalidad de la programación convexa disciplinada gracias a que, como hemos explicado antes, la librería es extensible.

3.1. Conjunto de reglas de convexidad

El conjunto de reglas de convexidad establece como las variables, parámetros, funciones y conjuntos pueden ser combinados para formar DCPs. Los DCPs son un subconjunto estricto de los CPs generales, por lo que otra manera de decir esto es que el conjunto de reglas impone un conjunto de convenciones o restricciones a los CPs. El conjunto de reglas se puede dividir en cuatro categorías: reglas de nivel superior, reglas sin productos, reglas de signo y reglas de composición.

Antes de enunciar las reglas, vamos a definir dos conceptos que utilizaremos, expresión constante y aserción.

Una expresión constante es una expresión que involucra solo valores numéricos y / o parámetros mientras que una expresión no constante depende del valor de al menos una variable del problema. Obviamente una expresión constante es trivialmente afín, convexa y cóncava. Estas expresiones deben estar bien definidas, es decir, que se verifiquen para cualquier conjunto de parámetros que satisfagan las aserciones del problema.

Una aserción se asemeja a una restricción, pero contiene solo expresiones constantes, está determinada completamente por los valores numéricos proporcionados para los parámetros de un modelo antes del comienzo de cualquier algoritmo de optimización numérico. Las aserciones sirven como condiciones previas, garantizando que un problema es numéricamente válido o físicamente significativo. Hay varias razones por las cuales se puede necesitar una aserción; por ejemplo:

- Representar los límites físicos dictados por el modelo. Por ejemplo, si un parámetro w representa el peso físico de un objeto, una afirmación $w > 0$ impone el hecho de que el peso debe ser positivo.

- Para asegurar que está bien definido. Por ejemplo, si x, y, z son variables y a, b, c son parámetros, entonces la restricción de desigualdad $ax + by + z/c \leq 1$ está bien planteada solo si c es distinta de cero; esto puede ser asegurado por una aserción como $c \neq 0$ o $c > 0$.

- Para garantizar el cumplimiento de las condiciones previas adjuntas a una función o conjunto en la librería. Por ejemplo, una función $f(x) = \|x\|_p$, está parametrizada por un valor $p \geq 1$. Si p se proporciona como un parámetro, entonces se requeriría una aserción como $p \geq 1$ para garantizar que la función se utiliza correctamente.

- Para asegurar el cumplimiento con las reglas de signo o las reglas de composición que veremos a continuación.

3.1.1. Reglas de nivel superior

Como su nombre indica, las reglas de nivel superior establecen la estructura superior de los DCPs. Estas reglas son más descriptivas que restrictivas, en el sentido de que casi todos los problemas convexos siguen estas convenciones, pero se deben explicitar.

Tipos de problemas. Un DCP válido puede ser:

- T1 un problema de minimización: un objetivo convexo y cero o más restricciones convexas.

- T2 un problema de maximización: un objetivo cóncavo y cero o más restricciones convexas.
- T3 un problema de factibilidad: no hay objetivo, y una o más restricciones convexas.

Un DCP válido también puede incluir cualquier número de aserciones.

Restricciones. Las restricciones válidas incluyen:

- T4 una restricción de igualdad con expresiones afines a izquierda y derecha.
- T5 una desigualdad menor o menor o igual que ($<$, $\leq$), con una expresión convexa a la izquierda y una expresión cóncava a la derecha.
- T6 una desigualdad mayor 0 mayor o igual que ($>$,$\geq$), con una expresión cóncava a la izquierda y una expresión convexa a la derecha.
- T7 una restricción de pertenencia establecida $(lexp_1, ..., lexp_m) \in cset$, donde $m \geq 1$, $lexp_1, ..., lexp_m$ son expresiones afines, y cset es un conjunto convexo.

Las restricciones de no igualdad ($\neq$) y las restricciones de no pertenencia no están permitidas, porque son convexas solo en casos excepcionales, y la filosofía de la programación convexa disciplinada no considera casos especiales. Sin embargo, en las aserciones estas dos operaciones pueden ser empleadas libremente ya que las aserciones no están restringidas de la manera en que lo están las restricciones reales.

Expresiones y afirmaciones constantes.

- T8 Cualquier expresión bien definida que conste únicamente de valores numéricos y parámetros es una expresión constante válida.
- T9 Cualquier expresión booleana que realice pruebas o comparaciones en expresiones constantes válidas es una aserción válida.
- T10 Si una función o conjunto está parametrizado, entonces esos parámetros deben ser expresiones constantes válidas.

Si consideramos por ejemplo esta función y conjunto:

$$\begin{array}{l} f_p : \mathbb{R}^n \to \mathbb{R}, \quad f_p(x) = \| \ x \ \|_p \\ B_p = \{x \in \mathbb{R}^n \mid \ \| \ x \ \|_p \leq 1\} \end{array} \tag{3.1}$$

Esta última regla simplemente impone que los parámetros como p deben ser constantes. Por supuesto esto es generalmente asumido, pero debemos explicitarlo.

3.1.2. Reglas libres de productos

La convención más destacada impuesta por la programación convexa disciplinada es la prohibición de productos entre expresiones no constantes, lo que incluye además operaciones relacionadas, como la exponencial. La necesidad de esta imposición se debe a que no existe un principio en el análisis convexo que pueda determinar la curvatura en tal caso. Podemos observarlo a partir de ejemplos. Supongamos que x es una variable escalar, entonces:

- La expresión $x \cdot x$, un producto de dos expresiones afines, es convexa.

- La expresión $x \cdot logx$, un producto entre una expresión afín y una expresión cóncava, es convexa.

- La expresión $x \cdot e^x$, un producto entre una expresión afín y una expresión convexa, no es convexa ni cóncava.

Por otro lado, a partir de los principios más básicos del análisis convexo tenemos las siguientes afirmaciones, probadas en el capítulo anterior:

- La suma de dos o más expresiones convexas (cóncavas, afines) es convexa (cóncava, afín).
- El producto de una expresión convexa (cóncava) y de una expresión constante no negativa es convexa (cóncava).
- El producto de una expresión convexa (cóncava) y una expresión constante no positiva, o la simple negación de la primera, es cóncava (convexa).
- El producto de una expresión afín y cualquier constante es afín.

Transformamos estos principios en un conjunto de reglas y como resultado tenemos las reglas, apropiadamente llamadas, libres de productos:

Reglas libre de productos: todas las expresiones numéricas válidas deben estar libres de productos; tales expresiones incluyen:

- PN1 Una variable simple.
- PN2 Una expresión constante.
- PN3 Una llamada a una función en la librería. Cada argumento de la función debe ser una expresión libre de producto.
- PN4 La suma de dos o más expresiones libres de productos.
- PN5 La diferencia de expresiones libres de producto.
- PN6 La negación de una expresión libre de producto.
- PN7 El producto de una expresión libre de producto y una expresión constante.
- PN8 La división de una expresión libre de producto por una expresión constante.

Asumimos en cada una de estas reglas que los resultados están bien definidos; por ejemplo, que las dimensiones son compatibles.

En el caso escalar, un modo de reformular estas reglas es decir que una expresión numérica válida puede reducirse a la forma:

$$a + \sum_{i=1}^{n} b_i x_i + \sum_{j=1}^{L} c_j f_j(arg_{j,1}, arg_{j,2}, ..., arg_{j,m_j}) \tag{3.2}$$

donde a, b_i, c_j son constantes; x_i son las variables problema; y $f_j : \mathbb{R}^{m_j} \to \mathbb{R}$ son funciones de la librería, y sus argumentos $arg_{j,k}$ son expresiones libres de productos. Podemos observar ciertos casos especiales:

1. Si $L = 0$ es una expresión afín simple.
2. Si, además, $b_1 = b_2 = ... = b_n = 0$, entonces es una expresión constante.

Para una ilustración del uso de estas reglas libres de productos, supongamos que a, b y c son parámetros; x, y, z son variables; y $f(\cdot)$, $g(\cdot,\cdot)$ y $h(\cdot,\cdot,\cdot)$ son funciones de la librería. Entonces la expresión:

$$af(x) + y + (h(x, bg(y, z), c) - z + b)/a$$

satisface la regla libre de producto, puede ser reescrita siguiendo forma 3.2:

$$(b/a) + y + (-1/a)z + af(x) + (1/a)h(x, bg(y, z), c)$$

Por otro lado, la expresión siguiente:

$$axy/2 - f(x)g(y, z) + h(x, y^b, z, c)$$

no se puede escribir como 3.2, hay un producto entre dos variables (xy) y un producto entre dos funciones (f(x)g(y,z)).

Para expresiones de conjuntos, las reglas libres de producto son similares. Utilizamos también las operaciones vistas en el capítulo anterior.

Reglas libres de producto para expresiones de conjunto: todas las expresiones de conjunto válidas usadas en restricciones deben estar libres de producto; tales expresiones incluyen:

- PSI Una llamada a un conjunto convexo en la librería.
- PS2 Una llamada a una función en la librería. Cada argumento de la función debe ser una expresión de conjunto libre de producto o una expresión constante (numérica).
- PS3 La suma de conjuntos libres de producto, o de un conjunto libre de producto y una expresión constante, o viceversa.
- PS4 La diferencia de un conjunto libre de productos y una expresión constante, o viceversa.
- PS5 La negación de un conjunto libre de productos.
- PS6 El producto de un conjunto libre de productos y una expresión constante.
- PS7 La división de un conjunto libre de producto por una expresión constante.
- PS8 La intersección de dos o más conjuntos libres de productos.
- PS9 El producto cartesiano de dos o más conjuntos libres de productos.

También asumimos en cada una de estas reglas que los resultados están bien definidos; por ejemplo, que las dimensiones son compatibles.

En otras palabras llamamos conjuntos válidos a expresiones que se pueden reducir a la forma:

$$a + \sum_{i=1}^{n} b_i S_i + \sum_{j=1}^{L} c_j f_j(arg_{j,1}, arg_{j,2}, ..., arg_{j,m_j}) \tag{3.3}$$

donde a, b_i, c_j son constantes, y S_k son conjuntos de la librería, o intersecciones y / o productos cartesianos de conjuntos válidos. Las funciones $f_j : \mathbb{R}_j^m \to \mathbb{R}$ son funciones de la librería, y sus argumentos $arg_{j,k}$ son expresiones de conjuntos libres de productos.

Las expresiones de conjuntos están más restringidas que las expresiones numéricas, ya que las funciones f_j deben ser afines, lo veremos a continuación.

Hemos visto en el capítulo anterior que la suma, intersección y producto cartesiano de conjuntos convexos es convexa; mientras que las uniones y las diferencias de conjuntos generalmente no son convexas. Lo que puede no estar claro es por qué las PS8-PS9 se consideran reglas "sin productos". Al examinar estas reglas en términos de funciones de indicadores, el enlace se vuelve claro. Consideremos, por ejemplo, el problema.

$$\begin{array}{ll} \text{Minimizar} & ax + by, \\ \text{Sujeto a:} & (x, y) \in \ (S_1 \times S_2) \cup S_2 \end{array}$$

si ϕ_1 y ϕ_2 son funciones indicador convexas para los conjuntos S_1 y S_2 respectivamente, es decir son de la forma:

$$\phi_i(x) = \begin{cases} 0 & si \quad x \in S_i \\ \\ \infty & si \quad x \notin S_i \end{cases}$$

entonces el problema se puede reducir a:

$$\text{Minimizar} \quad ax + by + (\phi_1(x) + \phi_2(y))\phi_2(x, y)$$

El producto escalar se ha convertido en la suma de las funciones indicador y la unión se ha convertido en un producto. Ahora la función objetivo viola la regla libre de producto debido a la operación generada por la unión, luego no se permite.

Si consideramos ahora el problema:

$$\begin{array}{ll} \text{Minimizar} & ax \\ \text{Sujeto a:} & x \in \ S_1 \cap S_2 \end{array}$$

se puede reducir a:

$$\text{Minimizar} \quad ax + \phi_1(x) + \phi_2(x)$$

La intersección se ha convertido también en la suma de las funciones indicador y por tanto es válida.

3.1.3. Reglas de signo

Las reglas de signo se denominan de esta manera porque su objetivo es determinar el signo de $c_1...c_L$ en 3.2 . Son un conjunto de reglas que establecen las condiciones suficientes para determinar si las expresiones son convexas, cóncavas o afines.

Podemos establecer de manera concisa las reglas de signo para las expresiones numéricas de la siguiente manera.

Reglas de signo para expresiones numéricas. Dada una expresión libre de producto, lo siguiente debe ser cierto en su forma reducida 3.2:

- SN1 Para que la expresión 3.2 sea convexa, cada término $c_j f_j(...)$ debe ser convexo; por lo tanto, uno de los siguientes puntos debe ser cierto:
  - $f_j(arg_{j,1}, ..., arg_{j,m_j})$ es afín.
  - $f_j(arg_{j,1}, ..., arg_{j,m_j})$ es convexo y $c_j \geq 0$.
  - $f_j(arg_{j,1}, ..., arg_{j,m_j})$ es cóncavo y $c_j \leq 0$.
- SN2 Para que la expresión 3.2 sea cóncava, cada término $c_j f_j(...)$ debe ser cóncavo; por lo tanto, uno de los siguientes puntos debe ser cierto:
  - $f_j(arg_{j,1}, ..., arg_{j,m_j})$ es afín.
  - $f_j(arg_{j,1}, ..., arg_{j,m_j})$ es cóncavo y $c_j \geq 0$.
  - $f_j(arg_{j,1}, ..., arg_{j,m_j})$ es convexo y $c_j \leq 0$.
- SN3 Para que la expresión 3.2 sea afín, cada función f_j debe ser afín, al igual que cada uno de sus argumentos $arg_{j,1}, ..., arg_{j,m_j}$.
- SN4 Para que la expresión 3.2 sea constante, entonces se debe cumplir que $L = 0$ y $b_1 = b_2 = ... = b = 0$.

Todos los argumentos de las funciones deben obedecer estas reglas también, según su curvatura esperada dictada por las reglas de composición.
Por ejemplo, supongamos que se espera que la expresión

$$af(x) + y + (h(x, bg(y, z), c) - z + b)/a$$

sea convexa y que la librería indica que la función $f(\cdot)$ es convexa, $g(\cdot, \cdot)$ es cóncava y $h(\cdot, \cdot, \cdot)$ es convexa. Entonces la regla del signo dicta que:

- Para que $af(x)$ sea convexa se debe tener $a \geq 0$
- Para que $(1/a)h(x, bg(y, z), c)$ sea convexa se debe tener $1/a \geq 0$
  Además se requiere que el segundo argumento de h sea convexo, es decir, $bg(y, z)$ debe ser convexo, esto implica que $b \leq 0$

Es responsabilidad del modelador asegurarse de que los valores de los coeficientes $c_1, ..., c_L$ obedecen las reglas de signo. Esto se puede lograr agregando las aserciones apropiadas al modelo.

Solo hay una regla de signo para las expresiones de conjuntos:

Regla de signos para conjuntos. Dada una expresión de conjunto libre de producto, lo siguiente debe ser cierto para su forma reducida 3.3:

SS1 Cada función f_j, y cualquier función utilizada en sus argumentos, debe ser afín.

Por ejemplo, para una expresión en la forma reducible con $L = 0$,

$$x \in a + \sum_{i=0}^{n} b_i S_i \quad \leftrightarrow$$
$$\exists (t_1, t_2, ..., t_n) \in S_1 \times S_2 \times ... \times S_n \text{ tal que } \quad x = a + \sum_{i=0}^{n} b_i t_i$$

Cuando $L > 0$, también se realizan sustituciones similares de forma recursiva en los argumentos de la función, produciendo un resultado similar: una serie de restricciones de pertenencia a conjuntos simples de la forma $t_k \in S_k$, y una restricción de igualdad única. Por lo tanto, para asegurar que esta restricción de igualdad implícita sea convexa, las expresiones de conjuntos (específicamente, aquellas usadas en restricciones) deben reducirse a combinaciones afines de conjuntos. Por supuesto, las expresiones de conjuntos utilizadas en las aserciones no están restringidas de esta manera.

3.1.4. Reglas de composición

Un principio básico del análisis convexo es que la composición de una función convexa con una transformación afín sigue siendo convexa. De hecho, bajo ciertas condiciones, también es cierto para composiciones con transformaciones no lineales. Se han planteado estos resultados en el capítulo anterior.

Supongamos que tenemos dos funciones convexas f, g y queremos realizar una composición. Por el resultado visto en el capítulo 2, sabemos que para asegurar que $f \circ g$ es convexa necesitamos que f sea no decreciente.

Se puede generalizar este razonamiento a funciones con múltiples argumentos, creando un conjunto de reglas denominadas reglas de composición.

Las reglas de composición. Considera una expresión numérica de la forma

$$f(arg_{j,1}, ..., arg_{j,m_j}) \tag{3.4}$$

donde f es una función de la librería. La expresión debe satisfacer exactamente una de las siguientes reglas:

- C1-C3 Para que la expresión sea convexa, entonces f debe ser afín o convexa, y una de las siguientes afirmaciones debe ser cierta para cada $k = 1, ..., m$:
  - C1 f es no decreciente en el argumento k y arg_k es convexo.
  - C2 f es no creciente en el argumento k y arg_k es cóncavo.
  - C3 arg_k es afín.
- C4-C6 Para que la expresión sea cóncava, entonces f debe ser afín o cóncavo, y una de las siguientes afirmaciones debe ser cierta para cada $k = 1, ..., m$:
  - C4 f es no decreciente en el argumento k y arg_k es cóncavo.
  - C5 f es no creciente en el argumento k y arg_k es convexo.
  - C6 arg_k es afín.
- C7 Para que la expresión sea afín, entonces f debe ser afín, y cada argumento es afín para todo $k = 1, ..., m$.

Estas reglas pueden ser demasiado conservativos y no dar como válida una composición que si lo sea. Es un pequeño precio que hay que pagar.

3.2. Verificación

Para resolver un problema mediante programación convexa disciplinada (DCP) el primer paso será la verificación, es decir, comprobar que es un problema convexo disciplinado válido, probar que incluye solo funciones y conjuntos de la librería y los combina de acuerdo al conjunto de reglas. La prueba de validez es de naturaleza jerárquica, refleja la estructura del problema y sus expresiones. Para ilustrar el proceso, consideremos el siguiente problema de optimización:

$$\begin{array}{lrl} \text{Minimizar} & e^x + y & \\ \text{Sujeto a:} & ax^2 - 3 & \leq \sqrt{y} \\ & log(x+y) & \geq 3 \end{array} \tag{3.5}$$

donde x, y son variables.

Vamos a dividir la prueba en dos etapas:

En la primera etapa se verificará si cada una de las expresiones del problema está libre de productos. Vamos a representar la jerarquía de la prueba, además de indicar la regla empleada en cada una de ellas.

- $e^x + y$ es libre de productos, por (PN4)
  - e^x es libre de productos (PN3)
    - x es libre de productos (PN1)
  - y es libre de productos (PN1)
- $ax^2 - 3$ es libre de producto, por (PN5)
  - ax^2 es libre de producto (PN7)
    - a es libre de producto, por (PN2)
    - x^2 es libre de producto, por (PN3)
      - x es libre de producto, por (PN1)
  - 3 es libre de producto, por (PN2)
- $\sqrt{y}$ es libre de producto, por (PN3)
  - y es libre de producto, por (PN1)
- $log(x+y)$ es libre de producto (PN3)
  - $x + y$ es libre de producto (PN4)
    - x es libre de producto, por (PN1)
    - y es libre de producto, por (PN1)
- 3 es libre de producto, por (PN2)

En la segunda etapa se verificará si el problema cumple las reglas de nivel superior, signo y composición. Lo representaremos de la misma forma.

Como veremos a continuación, el problema de minimización es válido si $a \geq 0$ por (T1)

La función objetivo es válida, por (T1)

- $e^x + y$ es convexa (SN1)
  - e^x es convexa, por (c1)
    - $e^{\cdot}$ es convexa, no decreciente (librería)
      - x es convexa, (SN1)
  - y es convexa, (SN1)

La primera restricción es válida si $a \geq 0$, por (T1)

- $ax^2 - 3 \leq \sqrt{y}$ es valida si $a \geq 0$, por (T5)
  - $ax^2 - 3$ es convexa si $a \geq 0$, por (SN1)
    - ax^2 es convexa si $a \geq 0$, por (SN1)
      - x^2 es convexa, por(C1)
      - square(.) es convexa, creciente (librería)
      - x es convexa, (SN1)
    - -3 es convexa, (SN1)
  - $\sqrt{y}$ es cóncava, por (C4)
    - $\sqrt{\cdot}$ es cóncava y no decreciente (librería)
    - y es cóncava (SN2)

La segunda restricción es válida, por (T1)

- $log(x + y) \geq 3$ es válida por(T6)
  - $log(x + y)$ es cóncavo, (C4)
    - $x + y$ es cóncava (SN2)
      - x es cóncava, (SN2)
      - y es cóncava, (SN2)
  - 3 es convexa, (SN1)

También es posible examinar gráficamente la estructura del problema y su validez mediante un árbol de expresiones del problema. Cada hoja representaría una operación o variable y se incluyen también las reglas usadas en cada paso.

El proceso de verificación garantiza una de estas tres conclusiones:

- Válido: las reglas están totalmente satisfechas.
- Condicionalmente válido: las reglas se cumplirán completamente si se cumplen una o más condiciones previas adicionales en los parámetros.
- No válido: una o más de las reglas de convexidad han sido violadas.

En este ejemplo, llegamos a la conclusión de que es condicionalmente válido: a través del análisis hemos visto que debe cumplirse una condición adicional, $a \geq 0$. Si esta condición previa estuviera asegurada de alguna manera, entonces la prueba habría determinado de manera concluyente que el problema es DCP válido. Esto se puede lograr agregando la aserción $a \geq 0$ a la lista de restricciones. Si, por otro lado, tuviéramos que agregar una aserción $a < 0$, se violaría la regla de signos SN2.

3.3. Crear programas convexos disciplinados

Como hemos visto, verificar el conjunto de reglas de convexidad es condición suficiente, no necesaria, para asegurar que el problema sea convexo. Un ejemplo muy común e importante son las formas cuadráticas, sabemos que son convexas y sin embargo no verifican las reglas libres de productos. Para poder resolver problemas con formas cuadráticas se incluye en la librería la función square(x) que significa x^2.

Es bastante sencillo encontrar problemas que son convexos pero no DCPs. Si un problema no verifica las reglas de convexidad no significa que no se pueda resolver en el marco de programación convexa disciplinada, a veces simplemente hay que reescribirlo.

Por ejemplo, vamos a considerar el problema de maximizar la entropía:

$$\begin{array}{rr} \text{Maximizar} & -\sum_{i=1}^{n} x_i log x_i \\ \text{Sujeto a:} & Ax = b \\ & 1^T x = 1 \\ & x \geq 0 \end{array} \tag{3.6}$$

donde $x \in \mathbb{R}^n$ es la variable del problema y $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ son parámetros; y $log(.)$ está definida en la librería. La expresión $x_i log x_i$ no verifica la regla libre de productos PN7, ya que contiene un producto entre dos expresiones no constantes. Luego este problema no es DCP válido, sin embargo sabemos que es un problema convexo.

La forma más sencilla de resolverlo en el marco de la programación convexa disciplinada es agregando una nueva función a la librería. En este caso necesitaríamos añadir la siguente función: $entr(x) = -x log x$ para $x > 0$. Es una función cóncava, con signo y monotonía desconocidos. A partir de esta función, el problema se puede reescribir de la siguiente manera como un problema dDCP válido:

$$\begin{array}{rr} \text{Maximizar} & -\sum_{i=1}^{n} entr(x_i) \\ \text{Sujeto a:} & Ax = b \\ & 1^T x = 1 \\ & x \geq 0 \end{array} \tag{3.7}$$

Una vez que se define e implementa una función, se puede reutilizar libremente para otros problemas y se pueden compartir con otros usuarios. El esfuerzo invertido en agregar una nueva función a la librería se amortiza significativamente.

3.4. Implementación de las funciones básicas

La implementación es una descripción fácil de usar de una función o conjunto que permite su uso en algoritmos de búsqueda de solución numérica óptima. Es importante tener en cuenta que no se utiliza para determinar si un problema en particular es o no DCP válido, entra en juego solo después de la verificación, al calcular una solución numérica.

Existen una gran variedad de métodos o solvers que se pueden emplear para resolver problemas convexos: métodos primal/dual, métodos de barrera, métodos de plano de corte, etc. No los podemos tratar todos en detalle en este trabajo, pero para más información se puede consultar ([33]). Todos estos métodos pueden adaptarse a la programación

disciplinada convexa con un mínimo esfuerzo. Cada uno de estos métodos necesitará realizar ciertos cálculos que involucren a cada una de las funciones y conjuntos, empleados en los problemas que resuelven. El objetivo de la implementación de una función es proporcionar a estos solvers los medios para realizar estos cálculos.

3.4.1. Implementaciones gráficas

Una implementación gráfica de una función es una representación del epigrafo o del hipografo de la función, como un problema convexo disciplinado factible.
El principal beneficio de las implementaciones gráficas es que proporcionan un elegante medio para definir funciones no diferenciables, por ejemplo la función valor absoluto o la función mínimo.

$f(x) = |x|$ es convexa, entonces se puede expresar como $f(x) = inf\{y \mid (x, y) \in Epi(f)\}$ con $Epi(f) = \{(x, y) \in \mathbb{R} \times \mathbb{R} \mid x \leq y, -x \leq y\}$

$g(x, y) = min\{x, y\}$ es cóncava, se puede expresar como $g(x, y) = sup\{z \mid (x, y, z) \in Hip(g)\}$ con $Hip(g) = \{(x, y, z) \in \mathbb{R} \times \mathbb{R} \times \mathbb{R} \mid z \leq x, z \leq y\}$

3.4.2. Uso de las implementaciones gráficas

Para resolver un DCP que involucre funciones o conjuntos con implementaciones gráficas, esas transformaciones deben aplicarse a través de un proceso que llamamos expansión gráfica, en el cual el DCP que describe una función dada de la librería se incorpora al problema. Para ilustrar lo que esto implica, consideremo el siguiente problema.

$$\begin{array}{rr} \text{Maximizar} & min\{c_1^T x + d_1, c_2^T x + d_2\} \\ \text{Sujeto a:} & Ax = b \\ & x \geq 0 \end{array} \tag{3.8}$$

Se emplea la función $min\{\cdot, \cdot\}$. Esta función es cóncava, hemos definido su hipografo en un ejemplo anterior.

Utilizando el hipografo, este problema se vuelve a escribir como:

$$\begin{array}{rr} \text{Maximizar} & sup\{y \mid y \leq c_1^T x + d_1, y \leq c_2^T x + d_2\} \\ \text{Sujeto a:} & Ax = b \\ & x \geq 0 \end{array} \tag{3.9}$$

Introduciendo la variable y en el modelo se obtiene el resultado expandido:

$$\begin{array}{rrl} \text{Maximizar} & y & \\ \text{Sujeto a:} & y \leq & c_1^T x + d_1 \\ & y \leq & c_2^T x + d_2 \\ & Ax = & b \\ & x \geq & 0 \end{array} \tag{3.10}$$

No es difícil ver que este problema es equivalente al original y, sin embargo, ahora es un LP simple.
Debido a que las implementaciones gráficas se expanden antes de implementar un algoritmo

numérico, no requieren ningún ajuste por parte de los algoritmos para admitirlas. Por lo tanto, las implementaciones gráficas son algoritmos independientes: cualquier algoritmo que pueda admitir funciones y conjuntos implementados mediante el cálculo de derivadas, sub / supergradientes, funciones de barrera, etc. también puede resolver problemas con funciones y conjuntos con implementaciones gráficas. Además, los algoritmos que no pueden admitir funciones no diferenciables si que pueden admitir su transformación mediante implementaciones gráficas.

Capítulo 4

CVXR

CVXR es un paquete de R que proporciona un lenguaje de modelado orientado a la resolución de problemas de programación convexa, similar a CVXPY en Pyton [20], Convex.jl en Julia [70], CVX [34] y YALMIP [46], ambos en Matlab.

R es un entorno y lenguaje de programación con un enfoque al análisis estadístico y gráfico. Fue desarrollado inicialmente por Robert Gentleman y Ross Ihaka del Departamento de Estadística de la Universidad de Auckland (Nueva Zelanda) en 1993. Se trata de uno de los lenguajes de programación más utilizados en investigación científica, siendo además muy popular en el campo de la minería de datos, la investigación biomédica, la bioinformática y las matemáticas financieras.
R al estar orientado a la Estadística, proporciona un amplio abanico de herramientas estadísticas y gráficas. Entre otras características, podemos nombrar su capacidad gráfica, que permite generar gráficos con alta calidad. R también puede usarse como herramienta de cálculo numérico y a la vez ser útil para la minería de datos.

CVXR resuelve una amplia clase de problemas de optimización convexa como mínimos cuadrados, regresión rigida y lasso, regresión isotónica, regresión de Huber y muchos otros modelos y métodos en estadística. Todos estos ejemplos, al menos en sus formas más básicas, están muy desarrollados y, por tanto, tienen paquetes R diseñados para ellos. Si utilizamos CVXR para resolver estos problemas funcionará, pero probablemente sea más lento que un algoritmo personalizado. Sin embargo, este no es el verdadero propósito de CVXR. Este paquete proporciona un lenguaje específico de dominio (DSL) que permite formular y resolver nuevos problemas para los cuales no existe un código personalizado. Es ideal para construir un modelo propio o uno que sea un refinamiento de un método existente. Su principal ventaja es la flexibilidad, con pocas líneas de código se puede transformar un problema y volverlo a resolver rápidamente. Su sintaxis es simple y matemáticamente intuitiva, se especifica un objetivo y un conjunto de restricciones mediante la combinación de constantes, variables y parámetros.
CVXR combina a la perfección con el código original de R y varios paquetes populares, lo que permite que se incorpore fácilmente en un marco analítico más amplio.

CVXR utiliza la programación convexa disciplinada para verificar que un problema es convexo. Para ello necesita conocer ciertas propiedades de las funciones que aparecen en el problema, como la curvatura y el signo. Para esto, se utilizan las funciones de una librería incluida en este paquete, donde todas las propiedades necesarias son conocidas. Una vez verificado, lo convierte a forma cónica estándar usando implementaciones gráficas

y lo resuelve pasándolo a un solver como ECOS [22] o SCS [55]. Un solver es una parte de un software matemático que resuelve un tipo de problema. Toma la descripción del problema en algún tipo de forma genérica y calcula su solución óptima.

Hemos hablado en el capítulo 3 que la programación convexa disciplinada se basa en una librería de funciones y conjuntos. También en cada una de las reglas hemos distinguido entre funciones y conjuntos. Sin embargo CVXR no posee librería de conjuntos, como tiene por ejemplo CVX en matlab, por lo que no podremos trabajar con ellos.

En este capítulo se explica como son aplicadas las reglas de la programación convexa disciplinada por CVXR y se describe su librería de funciones. La intención es mostrar el potencial de CVXR, para ello además expondremos una serie de aplicaciones prácticas. Podemos encontrar toda la información sobre este paquete de R en su página web oficial https://cvxr.rbind.io/ y en el artículo CVXR: An R Package for Disciplined Convex Optimization escrito por Anqi Fu, Balasubramanian Narasimhan y Stephen Boyd [26].

4.1. Expresiones

Las expresiones de CVXR están formadas por variables, constantes muméricas y matrices. Las operaciones aritméticas estándar son $+, -, *, \%*\%$, $/$, y la librería de funciones.

- Para crear variables o constantes utilizamos las órdenes Variable, Constant y matrix. En el siguiente código se ilustra su uso.

```
library(CVXR, warn.conflicts=FALSE)

# Variable unidimensional x
x <- Variable()

# Matriz de variables de 2 filas y tres columnas
y <- Variable(2,3)

# Constante 4
a <- Constant(4)

# Matriz de constantes de 3 filas y 2 columnas
B <- matrix(c(2,3,4,8,2,2), nrow=3)
```

Entre paréntesis indicamos las dimensiones de la variable, si no ponemos nada por defecto la crea con dimensión (1,1).

- Para determinar el signo y curvatura de cada expresión CVXR utiliza las reglas de programación convexa disciplinada de signo y composición, respectivamente.
  En el caso del signo, cada expresión se etiqueta como positiva, negativa, cero o desconocida (cuando el análisis DCP no puede determinar el signo).
  En el caso de la curvatura, cada expresión se etiqueta como constante, afín, cóncava, convexa o desconocida (cuando el análisis DCP no puede determinar la curvatura).

Podemos obtener el signo de cualquier expresión utilizando la función sign. Por ejemplo, utilizando los elementos definidos en el código anterior.

```
sign(y%*%B)
```

```
## [1] "UNKNOWN"
```

```
sign(a)
```

```
## [1] "POSITIVE"
```

Podemos obtener la curvatura de cualquier expresión utilizando la función curvature, por ejemplo:

```
curvature(sqrt(x))
```

```
## [1] "CONCAVE"
```

```
curvature(a)
```

```
## [1] "CONSTANT"
```

Para que un problema pueda ser resuelto por CVXR debe ser convexo. Entonces, dado un problema, el primer paso es verificar la convexidad. Para ello CVXR utiliza la programación convexa disciplinada, ya que si un problema sigue sus reglas podemos garantizar que es convexo.
Podemos conocer si un problema es convexo disciplinado o no mediante la función is_dcp. Vamos a mostrar algunos ejemplos:

$$\begin{array}{rl} \text{Minimizar} & (x-y)^2 \\ \text{Sujeto a:} & x+y \geq 0 \end{array}$$

En este caso, aunque el producto de dos variables no es DCP, la librería de funciones incluye square, lo que nos permite trabajar con variables elevadas al cuadrado.

```
# Definimos el problema:

x <- Variable()
y <- Variable()
prob1 <- Problem(Minimize(square(x - y)), list(x + y >= 0))

# Verificación
is_dcp(prob1)
```

```
## [1] TRUE
```

Veamos otro ejemplo.

$$\text{Maximizar} \quad e^x$$

En este ejemplo se maximiza una función convexa, no verifica la regla T2.

```
prob2 <- Problem(Maximize(exp(x)))

is_dcp(prob2)
```

```
## [1] FALSE
```

4.2. Funciones

Vamos a decribir las funciones que pueden ser aplicadas en CVXR.

Operadores

Los operadores $+, -, *, \%*\%$, / se tratan como funciones. + y - son funciones afines. $*$ y / son afines en CVXR porque expr1$*$expr2 y expr1 $\%*\%$ expr2 solo se permiten cuando una de las expresiones es constante y expr1 / expr2 solo se permite cuando expr2 es una constante escalar.

Indexación y corte

Todas las expresiones no escalares se pueden indexar utilizando la sintaxis "expr[i, j]". La indexación es una función afín. La sintaxis "expr[i]" se puede usar como una abreviatura para "expr[i, 1]" cuando expr es un vector columna. De manera similar, "expr[i]" es una abreviatura de "expr[1, i]" cuando expr es un vector fila.

Las expresiones no escalares también se pueden dividir utilizando la sintaxis de indexado de R estándar. Por ejemplo, "expr[i: j, r]" selecciona las filas i a j de la columna r y devuelve un vector.

CVXR admite la indexación avanzada utilizando listas de índices o matrices booleanas. La semántica es la misma que en R. Siempre que R devuelva un vector numérico, CVXR devuelve un vector columna.

Transponer

La transposición de cualquier expresión también es una función afín. Se puede obtener utilizando la sintaxis t(expr).

4.2.1. Funciones escalares

Una función escalar toma uno o más escalares, vectores o matrices como argumentos y devuelve un escalar.

Cuadro 4.1: Funciones escalares disponibles en el paquete CVXR

Función	**Significado**	**Dominio**	**Signo**	**Curvatura**	**Monotonía**
geo_mean(x,p) $p \in \mathbb{R}^n_+, p \neq 0$	$x_1^{1/n} \cdots x_n^{1/n}(x_1^{p_1} \cdots x_n^{p_n})^{\frac{1}{1^T p}}$	$x \in \mathbb{R}^n_+$	Positivo	Cóncava	$\uparrow$
harmonic_mean(x)	$\frac{n}{\frac{1}{x_1}+\ldots+\frac{1}{x_n}}$	$x \in \mathbb{R}^n_+$	Positivo	Cóncava	$\uparrow$
lambda_max(X)	$\lambda_{min}(X)$	$X \in \mathbb{S}^n$	Desconocido	Convexa	Ninguna
lambda_min(X)	$\lambda_{min}(X)$	$X \in \mathbb{S}^n$	Desconocido	Cóncava	Ninguna
lambda_sum_largest(X,k) k=1,...,n	suma de los k mayores autovalores de X	$x \in S^n$	Desconocido	Convexa	Ninguna
lambda_sum_smallest(X,k) k=1,...,n	suma de los k menores autovalores de X	$X \in \mathbb{S}^n$	Desconocido	Cóncava	Ninguna
log_det(X)	log(det(X))	$X \in \mathbb{S}^n_+$	Desconocido	Cóncava	Ninguna
log_sum_exp(X) $C \in \mathbb{R}^{p \times q}$	$log(\sum_{ij} e^{x_{ij}})$	$X \in \mathbb{R}^{m \times n}$	Desconocido	Convexa	$\uparrow$
matrix_frac(x,P)	$x^T P^{-1} x$	$X \in \mathbb{R}^n$ $P \in \mathbb{S}^n_{++}$	Positivo	Convexa	Ninguna
max_entries(X)	$max_{ij}\{X_{ij}\}$	$X \in \mathbb{R}^{m \times n}$	Signo de X	Convexa	$\uparrow$
min_entries(X)	$min_{ij}\{X_{ij}\}$	$X \in \mathbb{R}^{m \times n}$	Signo de X	Cóncava	$\uparrow$
mixed_norm(X,p,q)	$(\sum_k (\sum_l \lvert x_{k,l}\rvert^p)^{q/p})^{1/q}$	$X \in \mathbb{R}^{m \times n}$	Positivo	Convexa	Ninguna
cvxr_norm(x) cvxr_norm(x,2)	$\sqrt{\sum_i x_i^2}$	$x \in \mathbb{R}^n$	Positivo	Convexa	$\uparrow$ para $x_i \geq 0$, $\downarrow$ para $x_i \leq 0$
cvxr_norm(x,"fro")	$\sqrt{\sum_{ij} x_{ij}^2}$	$X \in \mathbb{R}^{m \times n}$	Positivo	Convexa	$\uparrow$ para $X_{ij} \geq 0$, $\downarrow$ para $X_{ij} \leq 0$
cvxr_norm(x,1)	$\sum_{ij} \lvert x_{ij}\rvert$	$X \in \mathbb{R}^{m \times n}$	Positivo	Convexa	$\uparrow$ para $X_{ij} \geq 0$, $\downarrow$ para $X_{ij} \leq 0$
cvxr_norm(x,"inf")	$max_{ij}\{\lvert x_{ij}\rvert\}$	$X \in \mathbb{R}^{m \times n}$	Positivo	Convexa	$\uparrow$ para $X_{ij} \geq 0$, $\downarrow$ para $X_{ij} \leq 0$
cvxr_norm(x,"nuc")	$tr((X^T X)^{1/2})$	$X \in \mathbb{R}^{m \times n}$	Positivo	Convexa	Ninguna
cvxr_norm(X) cvxr_norm(X,2)	$\sqrt{\lambda_{max}(X^T X)}$	$X \in \mathbb{R}^{m \times n}$	Positivo	Convexa	Ninguna
p_norm(X,p) $p \geq 1$ o $p = \infty$	$\lVert X \rVert_p = (\sum_{ij} \lvert x_{ij}\rvert^p)^{1/p}$	$X \in \mathbb{R}^{m \times n}$	Positivo	Convexa	$\uparrow$ para $X_{ij} \geq 0$, $\downarrow$ para $X_{ij} \leq 0$
p_norm(X,p) $p < 1$ o $p \neq 0$	$\lVert X \rVert_p = (\sum_{ij} x_{ij}^p)^{1/p}$	$X \in \mathbb{R}^{m \times n}_+$	Positivo	Cóncava	$\uparrow$
quad_form(x,P) $p \in \mathbb{S}^n_+$ *constante*	$x^T P x$	$X \in \mathbb{R}^n$	Positivo	Convexa	$\uparrow$ para $x_i \geq 0$, $\downarrow$ para $x_i \leq 0$
quad_form(x,P) $p \in \mathbb{S}^n_-$ *constante*	$x^T P x$	$X \in \mathbb{R}^n$	Negativo	Cóncava	$\uparrow$ para $x_i \geq 0$, $\downarrow$ para $x_i \leq 0$
quad_form(c,X) $c \in \mathbb{R}^n$ constante	$c^T X c$	$X \in \mathbb{R}^{n \times n}$	Depende de c,X	Afín	Depende de c
quad_over_lin(X,y)	$(\sum_{ij} x_{ij}^2)/y$	$x \in \mathbb{R}^n_+$	Positivo	Convexa	$\uparrow$ para $X_{ij} \geq 0$, $\downarrow$ para $X_{ij} \leq 0$ $\downarrow$ en y
sum_entries(X)	$\sum_{ij} x_{ij}$	$x \in \mathbb{R}^n_+$	Signo de X	Afín	$\uparrow$
sum_largest(X,k) $k = 1, 2, ...$	Suma de los k mayores X_{ij}	$X \in \mathbb{S}^n$	Signo de X	Convexa	$\uparrow$
sum_smallest(X,k) $k = 1, 2, ...$	Suma de los k menores X_{ij}	$X \in \mathbb{S}^n$	Signo de X	Cóncava	$\uparrow$
sum_squares(X)	$\sum_{ij} x_{ij}^2$	$X \in \mathbb{R}^{m \times n}$	Positivo	Convexa	$\uparrow$ para $X_{ij} \geq 0$, $\downarrow$ para $X_{ij} \leq 0$
matrix_trace(X)	tr(X)	$X \in \mathbb{S}^n$	Signo de X	Afín	$\uparrow$
tv(x)	$\sum_i \lvert x_{i+1} - x_i\rvert$	$X \in \mathbb{R}^n$	Positivo	Convexa	Ninguna
tv(X)	$\sum_{ij} \lVert \begin{bmatrix} X_{i+1,j} - X_{ij} \\ X_{i,j+1} - X_{ij} \end{bmatrix} \rVert_2$	$X \in \mathbb{R}^{m \times n}$	Positivo	Convexa	Ninguna
$tv(X_1, ..., X_k)$	$\sum_{ij} \lVert \begin{bmatrix} X^{(1)}_{i+1,j} - X^{(1)}_{ij} \\ X^{(1)}_{i,j+1} - X^{(1)}_{ij} \\ \vdots \\ X^{(k)}_{i+1,j} - X^{(k)}_{ij} \\ X^{(k)}_{i,j+1} - X^{(k)}_{ij} \end{bmatrix} \rVert_2$	$X^{(i)} \in \mathbb{R}^{m \times n}$	Positivo	Convexa	Ninguna

Aclaraciones

El dominio S^n se refiere al conjunto de matrices simétricas. Los dominios S^n_+ y S^n_- se refieren al conjunto de matrices semi-definidas positivas y semi-definidas negativas, respectivamente. De manera similar, S^n_{++} y S^n_{--} se refieren al conjunto de matrices definidas positivas y negativas, respectivamente.

Para una expresión vectorial x, "cvxr_norm(x)" y "cvxr_norm(x, 2)" dan la norma euclídea ($\| x \|_2 = \sqrt{x_1^2 + x_2^2 + ... + x_n^2}$), mientras que para una expresión matricial X "cvxr_norm(X)" y "cvxr_norm(X, 2)" dan la norma espectral ($\| A \|_2 = \sqrt{\lambda_{max(A^*A)}}$, donde A^* es la matriz traspuesta conjugada de A).

La función "cvxr_norm(X,"fro")" se denomina norma de Frobenius ($\| A \|_F = tr(A^*A)^{1/2} = (\sum_{i=1}^m \sum_{j=1}^n |A_{i,j}|^2)^{1/2}$) y cvxr_norm (X, "nuc") la norma nuclear($\| A \|_N = tr((A^T A)^{1/2})$). La norma nuclear también se puede definir como la suma de los valores singulares de X.

Las funciones max_entries y min_entries dan la entrada más grande y más pequeña, respectivamente, en una sola expresión. Estas funciones no deben confundirse con max_elemwise y min_elemwise que dan el máximo o el mínimo de una lista de expresiones escalares.

La función sum_entries suma todas las entradas en una sola expresión, mientras que "sum" suma una lista de expresiones.

Funciones a lo largo de un eje

Las funciones sum_entries, cvxr_norm, max_entries y min_entries se pueden aplicar a lo largo de un eje. Dada una expresión expr de dimensión (m,n) la sintaxis func(expr, axis = 1) aplica func a cada fila, devolviendo una expresión de dimensión (m,1). La sintaxis func(expr, axis = 2) aplica func a cada columna, devolviendo una expresión de dimensión (1,n).

En el siguiente ejemplo se construyen dos expresiones, en la primera se suman las variables de la matriz por filas y en la segunda por columnas.

```
library(CVXR, warn.conflicts=FALSE)

X <- Variable(5, 4)
row_sums <- sum_entries(X, axis=1) # Tiene dimensión (5, 1)
col_sums <- sum_entries(X, axis=2) # Tiene dimensión (1, 4)
```

4.2.2. Funciones element-wise

Estas funciones operan en cada elemento de sus argumentos. Por ejemplo, si X es una matriz de variables con dimensión (5,4), entonces "abs(X)" es una matriz de dimensión (5 ,4), es decir, "abs(X)[1, 2]" es equivalente a "abs(X [1, 2])".

Las funciones elementwise que toman múltiples argumentos, como "max_elemwise" y "mul_elemwise", operan en los elementos correspondientes de cada argumento. Por ejemplo, si X e Y son matrices de variables de dimensión (3,3), entonces "max_elemwise (X, Y)" es una matriz 3 por 3. "max_elemwise (X, Y) [2, 1]" es equivalente a "max_elemwise (X [2, 1], Y [2, 1])". Esto significa que todos los argumentos deben tener las mismas dimensiones o ser escalares.

Cuadro 4.2: Funciones element-wise disponibles en el paquete CVXR

Función	**Significado**	**Dominio**	**Signo**	**Curvatura**	**Monotonía**
abs(x)	$\mid x \mid$	$x \in \mathbb{R}$	Positivo	Convexa	$\uparrow$ para $x \leq 0$, $\downarrow$ para $x \geq 0$
entr(x)	$-xlog(x)$	$x > 0$	Desconocido	Cóncava	Ninguno
exp(x)	e^x	$x \in \mathbb{R}$	Positivo	Convexa	$\uparrow$
huber(x,M), $M \geq 0$	e^x	$x \in \mathbb{R}$	Positivo	Convexa	$\uparrow$ para $x \leq 0$, $\downarrow$ para $x \geq 0$
inv_pos(x)	$1/x$	$x > 0$	Positivo	Convexa	$\downarrow$
kl_div(x,y)	$xlog(x/y) - x + y$	$x > 0$, $y > 0$	Positivo	Covexa	Ninguno
log(x)	$log(x)$	$x > 0$	Desconocido	Cóncava	$\uparrow$
log1p(x)	$log(x + 1)$	$x > -1$	Signo de x	Cóncava	$\uparrow$
logistic(x)	$log(x + e^x)$	$x \in \mathbb{R}$	Positivo	Convexa	$\uparrow$
max_elewise(x_1, ..., x_k)	$max\{x_1, ..., x_k\}$	$x_i \in \mathbb{R}$	$max(sign(x_i))$	Convexa	$\uparrow$
min_elewise(x_1, ..., x_k)	$min\{x_1, ..., x_k\}$	$x_i \in \mathbb{R}$	$min(sign(x_i))$	Cóncava	$\uparrow$
mul_elewise(x_1, ..., x_k) $c \in \mathbb{R}$	$c * x$	$x \in \mathbb{R}$	$sign(cx)$	Afín	Depende de c
neg(x)	$max\{-x, 0\}$	$x \in \mathbb{R}$	Positivo	Convexa	$\downarrow$
pos(x)	$max\{x, 0\}$	$x \in \mathbb{R}$	Positivo	Convexa	$\uparrow$
power(x,0)	1	$x \in \mathbb{R}$	Positivo	Constante	constante
power(x,1)	x	$x \in \mathbb{R}$	La misma de x	Afín	$\uparrow$
power(x,p) p=2,4,8,...	x^p	$x \in \mathbb{R}$	Positivo	Convexa	$\uparrow$ para $x \leq 0$ $\downarrow$ para $x \geq 0$
power(x,p) p<0	x^p	$x > 0$	Positivo	Convexa	$\downarrow$
power(x,p) 0<p<1	x^p	≥ 0	Positivo	Cóncava	$\uparrow$
power(x,p) p>1 $p \neq 1, 4, 8, ...$	x^p	$x \geq 0$	Positivo	Convexa	$\uparrow$
$scalene(x, \alpha, \beta)$ $\alpha \geq 0$ $\beta \geq 0$	$\alpha pos(x) + \beta neg(x)$	$x \in \mathbb{R}$	Positivo	Convexa	$\uparrow$ para $x \leq 0$ $\downarrow$ para $x \geq 0$
sqrt(x)	$\sqrt{x}$	$x \geq 0$	Positivo	Cóncava	$\uparrow$
square(x)	x^2	$x \in \mathbb{R}$	Positivo	Convexa	$\uparrow$ para $x \leq 0$ $\downarrow$ para $x \geq 0$

4.2.3. Funciones vector / matriz

Una función de vector / matriz toma uno o más escalares, vectores o matrices como argumentos y devuelve un vector o matriz.

Cuadro 4.3: Funciones vector/matriz disponibles en el paquete CVXR

Función	Significado	Dominio	Signo	Curvatura	Monotonía
$bmat([[X_{11(},...,X_{1q}]$ $,...,[X_{q1},...,X_{qq}]])$	$\begin{bmatrix} X^{(1,1)} & ... & X^{(1,q)} \\ \vdots & & \vdots \\ X^{(p,1)} & ... & X^{(p,q)} \end{bmatrix}$	$x^{(i,j)} \in \mathbb{R}^{m_i \times n_j}$	$signo(\sum_{ij} X_{11}^{(i,j)})$	Afín	↑
conv(c,x) , $c \in \mathbb{R}$	c*x	$X \in \mathbb{R}^n$	$Signo(c_1 x_1)$	Afín	Depende de C
cumsum_axis(X,axis=1)	Suma acumulada a lo largo del eje dado	$X \in \mathbb{R}^{m \times n}$	Signo de X	Afín	↑
diag(x), $M \geq 0$	$\begin{bmatrix} X_1 & & \\ & \ddots & \\ & & X_n \end{bmatrix}$	$X \in \mathbb{R}^n$	Signo de X	Afín	↑
diag(X)	$\begin{bmatrix} X_1 \\ \vdots \\ X_n \end{bmatrix}$	$X \in \mathbb{R}^{n \times n}$	Signo de X	Afín	↑
diff(X,k=1,axis=1) $k \in 0, 1, 2, ...$	Diferencias de orden k (el argumento k se denomina diferencias y también se puede usar lag a lo largo del eje dado	$X \in \mathbb{R}^{m \times n}$	Signo de X	Afín	↑
$hstack(X_1, ..., X_k)$	$\begin{bmatrix} X^{(1)} & ... & X^{(k)} \end{bmatrix}$	$X^{(i)} \in \mathbb{R}^{m \times n_i}$	$signo(\sum_i X_{11}^{(i)})$	Afín	↑
kronecker(C,X) $C \in \mathbb{R}^{p \times q}$	$\begin{bmatrix} C_{11}X & ... & C_{1q}X \\ \vdots & & \vdots \\ C_{p1}X & ... & C_{pq}X \end{bmatrix}$	$X \in \mathbb{R}^{m \times n}$	$Signo(c_{11}x_{11})$	Afín	Depende de C
$reshape\_expr(X, n', m')$	X'$\in \mathbb{R}^{m' \times n'}$	$X \in \mathbb{R}^{m \times n}$ $m'n' = mn$	Signo de X	Afín	↑
vec(X)	x'$\in \mathbb{R}^{mn}$	$X \in \mathbb{R}^{m \times n}$	Signo de X	Afín	↑
$vstack(X_1, ..., X_k)$	$\begin{bmatrix} X^{(1)} \\ \vdots \\ X^{(n)} \end{bmatrix}$	$X^{(i)} \in \mathbb{R}^{m_i \times n}$	$signo(\sum_i X_{11}^{(i)})$	Afín	↑

Aclaraciones

La entrada de bmat es una lista de listas de expresiones CVXR y construye una matriz de bloques. Los elementos de cada lista interna se apilan horizontalmente, y luego las matrices de bloques resultantes se apilan verticalmente.

La salida de conv(c,x) es de tamaño n+m-1 y se define como $y[k] = \sum_{j=0}^{k} c[j]x[k-j]$.

La salida de vec (X) es la matriz X aplanada en orden de columna principal en un vector. Formalmente, $x'_i = X_{i \ mod \ m,[i/m]}$

La salida X de reshape_expr (X, m', n') es la matriz X convertida en una matriz m'xn'. Las entradas se toman de X por columnas y se almacenan en X' por columnas también. Formalmente, $X'_{ij} = vec(X)_{m'j+i}$.

4.3. Construir un problema

Empecemos con un problema simple. Sean x, y, z variables escalares, definimos:

$$\begin{array}{lrl} \text{Minimizar} & x + 2y - 0.1z & \\ \text{Sujeto a:} & 2x - y & \geq 0 \\ & -x + 3y & \geq 0 \\ & x + y & \leq 5 \\ & z & \geq 1.1 \\ & x, y, z & \geq 0 \\ & z & \in \mathbb{Z} \end{array} \tag{4.1}$$

Es un problema convexo, concretamente lineal, con $f(x, y) = x + 2y - 0.1z$ función objetivo y con las restricciones $f_1(x) = -2x + y$, $f_2(x, y) = x - 3y$, $f_3(x, y) = x + y - 5$, $f_4(x, y) = 1.1 - z$, $f_5(x, y) = -x$, $f_6(x, y) = -y$, $f_7(x, y) = -z$, $z \in \mathbb{Z}$.

Vamos a describir detalladamente el código CVXR necesario para resolverlo. Se carga la librería CVXR y empezamos por declarar las variables. Podemos imponer que alguna de las variables sea entera mediante Int():

```
library(CVXR, warn.conflicts=FALSE)

x <- Variable()
y <- Variable()
z <- Int()
```

Ahora se define el objetivo, se utiliza Maximize() para funciones cóncavas y Minimize() para funciones convexas. En este caso empleamos Maximize():

```
objetivo <- Maximize(x + 2*y - 0.1*z)
```

A continuación vamos a definir las restricciones de igualdad y desigualdad:

```
restricciones <- list(x>=0, y>=0, z>=0 , x+y<=5, 2*x-y>=0,
                      -x+3*y>=0,z>=1.1)
```

Entonces, utilizando el objetivo y las restricciones como objetos de entrada, ya podemos definir el problema mediante la función "Problem()":

```
problema <- Problem(objetivo,restricciones)
```

Los objetos problema son bastante flexibles, pueden tener restricciones o no.
Una vez definido el problema, comprobamos si es dcp.

```
is_dcp(problema)
```

```
## [1] TRUE
```

Por último, para resolver el problema se utiliza la función solve:

```
solucion <- solve(problema)
```

Esta llamada traduce el problema a un formato que un solver convexo puede entender, pasa el problema al solver y luego devuelve los resultados en una lista. Veamos como recuperar estos resultados:

```
solucion$status
```

```
## [1] "optimal"
```

Devuelve el estado del problema, que en este caso es óptimo por lo que devuelve "optimal". Es decir, se ha encontrado solución óptima para el problema. Se encontraron valores para las variables de decisión que satisfacen todas las restricciones y maximizan (en otros casos minimizan) el objetivo. En general, al aplicar el metodo solve() puede ocurrir que el problema sea óptimo como este, pero también nos podemos encontrar otros dos casos:

Si solucion$status == "infeasible", el problema no se ha resuelto porque no existe una combinación de variables de decisión que pueda satisfacer todas las restricciones.

Si solucion$estado == "unbounded": el problema no se resolvió porque el objetivo se puede hacer arbitrariamente pequeño para un problema de minimización o arbitrariamente grande para un problema de maximización. Por lo tanto, no hay una solución óptima porque, para cualquier solución dada, siempre es posible encontrar otra aún mejor.

```
solucion$value
```

```
## [1] 8.133333
```

Devuelve el valor óptimo de la función objetivo.
Si el problema es infactible devuelve +Inf en los problemas de mínimos y -Inf en los problemas de máximos.
Si el problema es ilimitado, devuelve -Inf para un problema de minimización y + Inf para un problema de maximización,

```
solucion$getValue(x)
```

```
## [1] 1.666667
```

```
solucion$getValue(y)
```

```
## [1] 3.333333
```

```
solucion$getValue(z)
```

```
## [1] 2
```

Devuelve los valores de las variables para los que se alcanza el óptimo. Si el problema es infactible o ilimitado devuelve NA.

4.3.1. Problema vectorial

Vamos a ver ahora como construir un problema con variables vectoriales. Consideremos el siguiente problema.

$$\begin{array}{rrl} \text{Minimizar} & \sum_{i=1}^{3} x_i & \\ \text{Sujeto a:} & Ax = & b \\ & x \geq & 0 \end{array}$$

con A=$\begin{pmatrix} 4 & 2 & 4 \\ 1 & 2 & 3 \\ 1 & 1 & 2 \end{pmatrix}$, b=$\begin{pmatrix} 4 \\ 4 \\ 4 \end{pmatrix}$.

El procedimiento será el mismo que en el caso anterior, empezamos declarando la variable.

```
library(CVXR, warn.conflicts=FALSE)

X <- Variable(3,1)
```

Ahora se define el objetivo, en este caso vamos a emplear "Minimize()" y la función de la librería "sum_entries()":

```
objetivoVec <- Minimize(sum_entries(X))
```

Se definen las restricciones:

```
A = matrix(c(4,2,4,1,2,3,1,1,2), nrow=3)
b = matrix(c(4,4,4), nrow=3)
restriccionesVec <- list(A%*%X==b, X>=0)
```

A partir del objetivo y las restricciones podemos definir el problema.

```
problemaVec <- Problem(objetivoVec,restriccionesVec)
```

Comprobamos que es dcp.

```
is_dcp(problemaVec)
```

```
## [1] TRUE
```

Por último, resolvemos mediante solve.

```
solucionVec <- solve(problemaVec)

solucionVec$status
```

```
## [1] "infeasible"
```

El estado del problema es infactible.

```
solucionVec$value
```

```
## [1] Inf
```

Como es un problema de mínimo y es infactible devuelve +Inf.

```
solucionVec$getValue(X)
```

```
## [1] NA
```

Vemos que al ser el problema infactible devuelve NA.

4.4. Implementación

CVXR representa las funciones, variables, restricciones y otras partes de un problema de optimización utilizando objetos R de clase S4. S4 nos permite sobrecargar operaciones matemáticas estándar para que CVXR se combine perfectamente con el script R original y otros paquetes. Cuando se invoca una operación en una variable, se crea un nuevo objeto que representa el árbol de expresión correspondiente con el operador como nodo raíz y

los argumentos como hojas. Este árbol crece automáticamente a medida que se agregan más elementos, lo que nos permite encapsular la estructura de una función objetivo o restricción.

Una vez que el usuario llama a la orden solve, ocurre la verificación DCP. CVXR recorre el árbol de la expresión de forma recursiva, determinando el signo y curvatura de cada subexpresión basándose en las propiedades de las funciones básicas que las componen. Si se considera que el problema se ajusta a las reglas, es transformado en un programa de cono equivalente utilizando implementaciones gráficas de funciones convexas. Luego, CVXR pasa la descripción del problema a la librería CVXcanon C ++ [49], que genera los datos para el programa de cono y envía estos datos a la interfaz de R específica del solver. Los resultados del solver se devuelven al usuario en una lista. Este diseño e infraestructura orientados a objetos de CVXR se tomaron en gran parte de CVXPY (versión escrita en lenguaje Phyton).

CVXR interactúa con los solvers de conos de código abierto ECOS [22] y SCS [55] a través de sus respectivos paquetes R. ECOS es un solver de punto interior, que logra una alta precisión para problemas pequeños y medianos, mientras que SCS es un solver de primer orden que es capaz de manejar problemas más grandes y restricciones semidefinidas. Ambos solvers ejecutan un solo hilo a la vez, el soporte para SCS multihilo se agregará en el futuro. Además, la versión 0.99 de CVXR proporciona soporte para los solvers comerciales MOSEK [4] y GUROBI [37] a través de paquetes binarios R publicados por los proveedores respectivos. No es difícil conectar solvers adicionales siempre que el solver tenga una API que pueda comunicarse con R. Los usuarios que deseen emplear un solver personalizado pueden obtener la combinación de los datos canonizados para un problema y solver directamente con get_problem_data (problema, solver). Cuando un problema puede ser resuelto por más de un solver, el argumento del solver de la función de resolución puede usarse para indicar una preferencia. Los solvers disponibles, según los paquetes instalados en una sesión, se devuelven a través de installed_solvers ().

Si ejecutamos dicha función,

```
installed_solvers ()
```

```
## [1] "ECOS"    "ECOS_BB" "SCS"
```

obtenemos los solvers mencionados, ECOS y SCS, y además aparece ECOS_BB, que es una extensión reciente de ECOS para resolver programas enteros mixtos o booleanos mixtos.

También podemos obtener qué solver ha utilizado para resolver un cierto problema. Podemos verlo utilizando el ejemplo aterior,

```
solucion$solver
```

```
## [1] "ECOS_BB"
```

Veamos ahora como podemos elegir el solver usado para resolver el problema,

Si queremos indicar ECOS_BB (en este caso es el que ha usado).

```
solucionECOS <- solve(problema,"ECOS_BB")
```

Si le indicamos que use, por ejemplo ECOS,

```
# solucionECOS <- solve(problema,"ECOS")
```

Nos devuelve error porque el solver ECOS no es válido para problemas mixtos enteros.

Además podemos pedir que nos devuelva el tiempo empleado en cargar y resolver el problema.

```
solucion$setup_time
```

```
## [1] 2.559999e-05
```

```
solucion$solve_time
```

```
## [1] 1.024e-05
```

Tenemos una gran librería de funciones que deberían ser suficiente para modelar la mayoría de los problemas de optimización convexa. Sin embargo, es posible añadir nuevas funciones a la librería. El proceso conlleva crear en R una clase S4 para la función, la sobrecarga de métodos que caracterizan sus propiedades DCP y representar su implementación gráfica como una lista de operadores lineales que especifican la factibilidad del problema.

4.5. Mínimos cuadrados

Para mostrar algunas de las ventajas de CVXR sobre otros paquetes de R vamos a tomar como ejemplo el problema de mínimos cuadrados, el cual puede ser resuelto en R utilizando lm.

$$\text{Minimizar} \quad \| y - X\beta \|_2^2 \tag{4.2}$$

Se traduce a CVXR directamente como se muestra a continuación:

```
# Generar los datos:

n <- 100
p <- 10
beta <- -4:5

x <- matrix(rnorm(n*p), nrow = n)
colnames(x) <- paste("beta_",beta)
y <- x%*%beta+rnorm(n)
```

```
# Resolver el problema usando CVXR

library(CVXR, warn.conflicts=FALSE)

# Definir la variable y el objetivo

betah <-Variable(p)
objetivo <- Minimize(sum((y-x%*%betah)^2))
```

Cuadro 4.4: Solución del problema de mínimos cuadrados mediante CVXR y lm.

	CVXR est.	lm est.
beta_1	-3.9207176	-3.9207179
beta_2	-2.9389253	-2.9389243
beta_3	-2.0783159	-2.0783182
beta_4	-1.0294941	-1.0294961
beta_5	-0.0299090	-0.0299107
beta_6	0.9346833	0.9346830
beta_7	1.9866211	1.9866201
beta_8	2.9100506	2.9100528
beta_9	3.8374584	3.8374583
beta_10	4.9595722	4.9595669

```
# Crear el problema

prob <- Problem(objetivo)
```

```
# Comprobar que el problema es DCP válido

is_dcp(prob)
```

```
## [1] TRUE
```

```
# Resolver

sol<-solve(prob)
```

Lo resolvemos también utilizando lm, así podremos comparar resultados.

```
lmmodelo <- lm(y~0+x)
```

En el cuadro 4.4 vemos resultados obtenidos. Podemos observar que en ambos casos el resultado es el mismo, pero aparentemente hemos reemplazado una llamada a lm por varias líneas. Entonces, ¿qué hemos ganado? Obviamente para resolver este problema deberíamos usar simplemente lm, la principal ventaja de CVXR es resolver problemas que no encajan en otros paquetes estándar. Por ejemplo si en este problema añadimos las restricciones $\beta_2 + \beta_3 \leq 0$ y $\beta_1, \beta_2, ..., \beta_{10} \geq 0$.

```
restriccion1 <- betah[2]+betah[3] <=0
B <- diag(c(1, 0, 0, rep(1, 7)))
restrccion2 <- B %*% betah >= 0

prob2 <- Problem(objetivo,  constraints = list(restriccion1,restrccion2))

sol2 <- solve(prob2)
```

En el cuadro 4.5 tenemos el resultado de este caso. Con este ejemplo podemos observar como se puede resolver un problema que no encaja en otro paquete de R, simplemente añadiendo las restricciones y como con pocas líneas de código se puede transformar un

Cuadro 4.5: Solución del probema de mínimos cuadrados con restricciones.

beta_1	0.00000
beta_2	-3.46311
beta_3	-1.86003
beta_4	0.00000
beta_5	0.02662
beta_6	0.78559
beta_7	2.44934
beta_8	2.29887
beta_9	3.97389
beta_10	5.43786

problema y volverlo a resolver rápidamente. CVRX es muy flexible y utiliza una sintaxis simple y matemáticamente intuitiva, su principal ventaja.

4.6. Aplicaciones

En este apartado vamos a exponer la resolución de algunas aplicaciones de la programación convexa mediante CVXR.

4.6.1. Problema de la Catenaria

Consideremos una versión discreta del problema de la catenaria en Griva and Vanderbei (2005) [36].

La catenaria es una curva ideal que representa físicamente la curva generada por una cadena, cuerda o cable sin rigidez flexional, suspendida de sus dos extremos y sometida a un campo gravitatorio uniforme.

En este caso vamos a considerar una cadena de longitud 2 cuya masa está uniformemente distribuida entre los puntos (0,1) y (1,1) en el plano. La fuerza gravitacional actúa en la dirección negativa de y. Nuestro objetivo es encontrar la forma de la cadena en equilibrio, lo que es equivalente a determinar las coordenadas (x, y) de cada punto a lo largo de la curva cuando la energía potencial es mínima.
Para formular esto como un problema de optimización parametrizamos la cadena según su longitud de arco y la dividimos en m segmentos. La longitud de cada segmento debe ser menor que un cierto $h > 0$. Ya que la masa es uniforme, la energía potencial total es simplemente la suma de las coordenadas y. Por lo tanto, nuestro problema es:

$$\begin{array}{lrl} \text{Minimizar} & \sum_{i=1}^{m} y_i & \\ \text{Sujeto a:} & x_1 & = 0 \\ & y_1 & = 1 \\ & x_m & = 1 \\ & y_m & = 1 \\ & (x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 & \leq h^2,\ i = 1, ..., m-1 \end{array} \tag{4.3}$$

con variables x, y. Este problema de catenaria básico tiene una solución analítica bien

conocida (Gelfand and Fomin 1963 [28]),

$$y(x) = acosh((x + C1)/a) + C2$$

a, C1, C2 son constantes determinadas por las resticciones que impongamos, a es el cociente entre el peso unitario de la cadena y la tensión.

Podemos verificar esta solución fácilmente con CVXR:

```
library(CVXR)

## Datos del problema

# Tomamos m+1 puntos para dividir a continuación la cuerda en
# m segmentos
m <- 101
# Se define la longitud de la cadena
L <- 2
# Se define la longitud de cada segmento
h <- L / (m - 1)

## Variables y objetivo del problema

x <- Variable(m)
y <- Variable(m)
objetivo <- Minimize(sum(y))


## Restricciones

restricciones <- list(x[1] == 0, y[1] == 1,
                      x[m] == 1, y[m] == 1,
                      diff(x)^2 + diff(y)^2 <= h^2)

prob <- Problem(objetivo, restricciones)
```

```
# Comprobar que el problema es DCP válido

is_dcp(prob)
```

```
## [1] TRUE
```

```
## Resolver

result <- solve(prob)
xs <- result$getValue(x)
ys <- result$getValue(y)

#Podemos ver que solver ha empleado
result$solver
```

```
## [1] "ECOS"
```

Cuadro 4.6: Primero valores de la solución del problema de la Catenaria.

x	y
0.0000000	1.0000000
0.0045205	0.9805116
0.0091287	0.9610469
0.0138282	0.9416054
0.0186229	0.9221879
0.0235166	0.9027957

En el cuadro 4.6 tenemos los primeros valores de la solución obtenida.

Ahora vamos a representar esta solución, obtenida mediante CVXR, y la solución analítica para poder comparar los resultados.

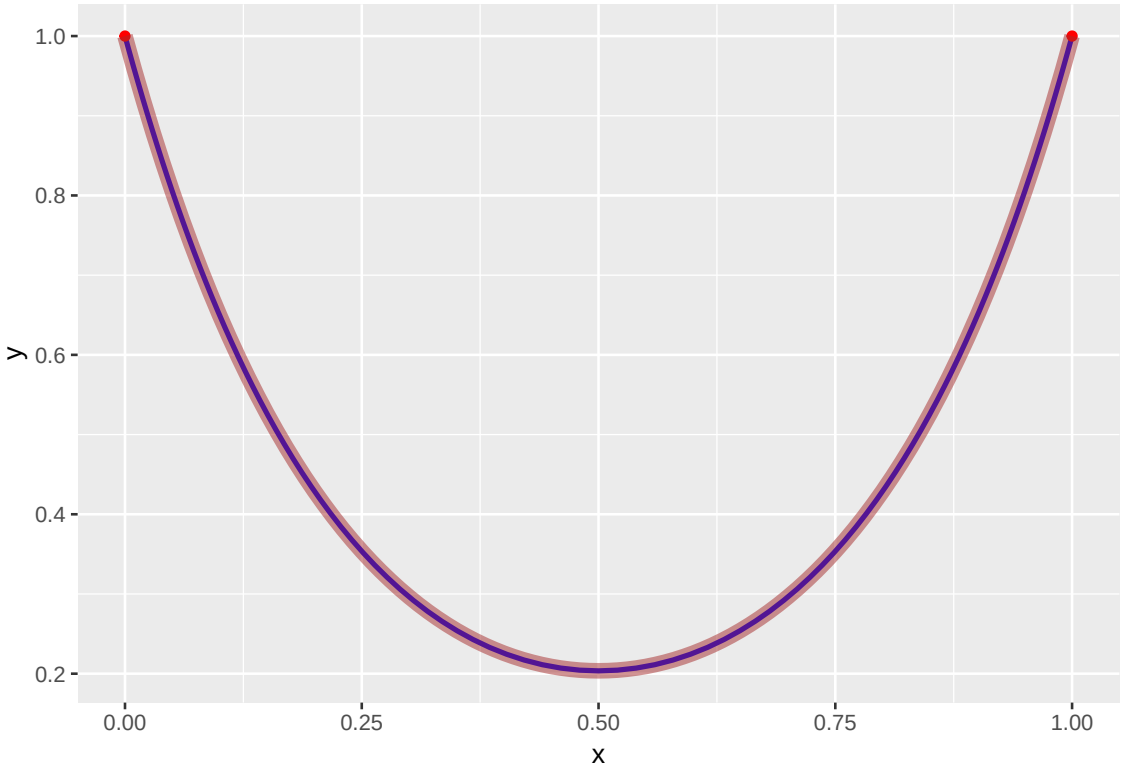

Figura 4.1: Representación gráfica de la solución al problema de la Catenaria mediante CVXR y solución analítica.

Podemos observar en la gráfica 4.1 como las soluciones obtenidas de ambas formas son exactamente iguales.

Masa no uniforme

Vamos a resolver el problema en el caso que la masa de la cadena no es uniforme. Suponemos que la masa es uno excepto en el nodo central y en los nodos que están a un cuarto de longitud de cada extremo. Ahora tenemos en cuenta la masa en la función objetivo, las restricciones se mantienen igual.

Cuadro 4.7: Primeros valores de la solución del probema de la catenaria con masa no uniforme.

x	y
0.0000000	1.0000000
0.0052373	0.9806948
0.0105265	0.9614038
0.0158683	0.9421276
0.0212639	0.9228669
0.0267144	0.9036216

```
# Dato de la masa

masa <- matrix(rep(1,m),nrow=m)
masa[m/4]=0.3*m;
masa[m/2]=0.3*m;
masa[3*m/4]=0.3*m;

# Definir el nuevo objetivo y problema

objetivo2 <- Minimize(sum(masa*y))
prob2 <- Problem(objetivo2, restricciones)
result2 <- solve(prob2)
xsm <- result2$getValue(x)
ysm <- result2$getValue(y)
```

En el cuadro 4.7 tenemos los primeros valores de la solución obtenida en este caso y en la gráfica 4.2 podemos observar su representación.

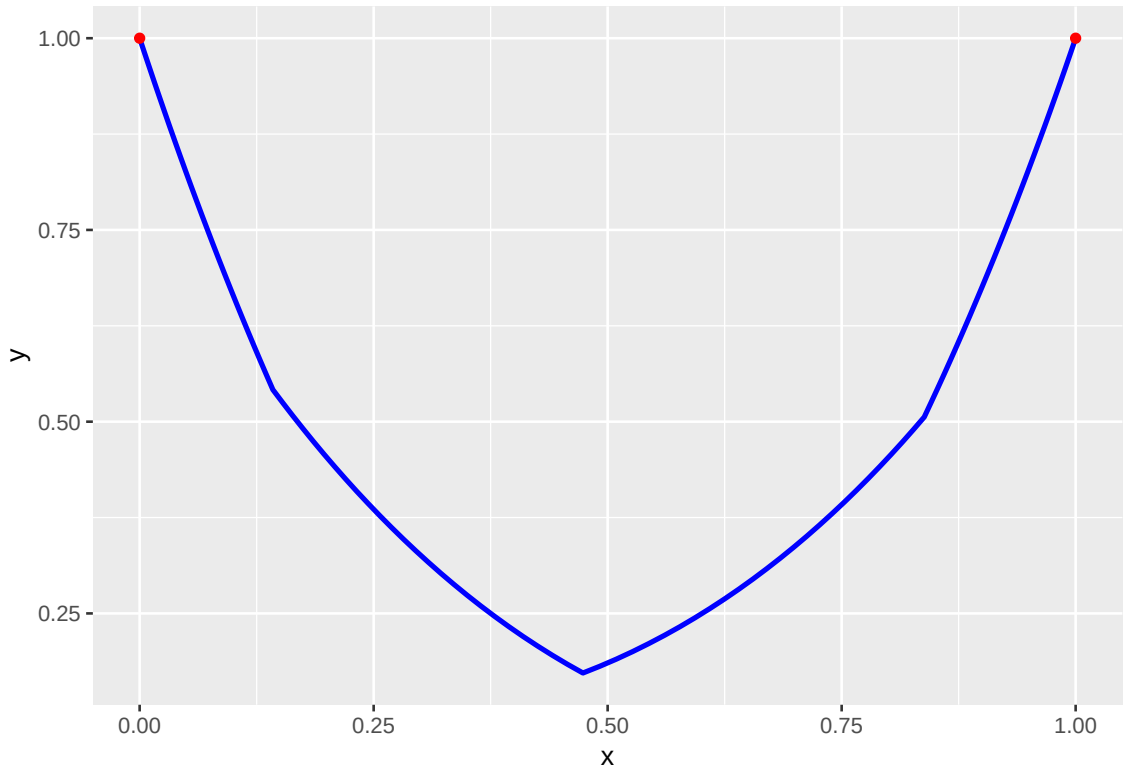

Figura 4.2: Representación gráfica del problema de la Catenaria con masa no uniforme.

Restricción sobre el suelo

Otra situación interesante puede ser cuando el suelo no es plano. Sea g el vector de elevación (relativo al eje x), y suponiendo que el punto final derecho de la cadena ha sido bajado 0.5. En este caso la solución analítica puede ser más difícil de calcular. Sin embargo, solo necesitamos añadir unas líneas a nuestra definición de las restricciones para obtener el resultado.

```
# Definir la estructura del suelo de escalera y bajar el punto extremo
# derecho

g <- sapply(seq(0, 1, length.out = m), function(x) {
    if(x < 0.2)
        return(0.6)
    else if(x >= 0.2 && x < 0.4)
        return(0.4)
    else if(x >= 0.4 && x < 0.6)
        return(0.2)
    else
        return(0)
})

# Definir el nuevo problema

restricciones3 <- c(restricciones, y >= g)
```

Cuadro 4.8: Primeros valores de la solución del probema de la catenaria con restricciones sobre el suelo.

x	y
0.0000000	1.0000000
0.0025235	0.9801594
0.0051234	0.9603286
0.0078044	0.9405084
0.0105718	0.9207003
0.0134311	0.9009051

```
restricciones3[[4]] <- (y[m] == 0.5)
prob3 <- Problem(objetivo, restricciones3)
result3 <- solve(prob3)
xs2 <- result3$getValue(x)
ys2 <- result3$getValue(y)
```

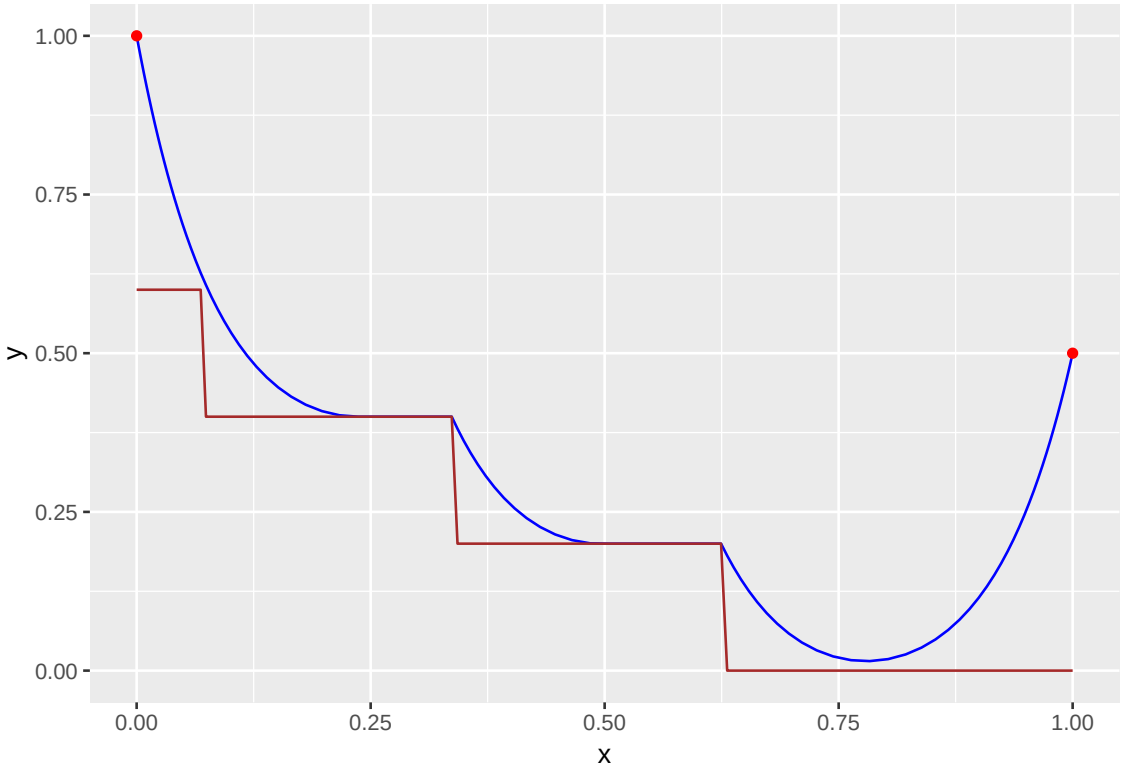

Figura 4.3: Representación gráfica del problema de la Catenaria con restricciones sobre el suelo.

En el cuadro 4.8 tenemos los primeros valores de la solución obtenida y en la gráfica 4.3 podemos observar su representación.

4.6.2. Regresión logística

Supongamos que queremos resolver un modelo de predicción para una variable binaria $y_i \in \{0, 1\}$, utilizamos la regresión logística (Cox 1958 [16]; Freedman 2009 [25]) . Modelamos la respuesta condicional como $y|x \sim Bernoulli(g_\beta(x))$, donde $g_\beta(x) = \frac{1}{1+e^{-x^T\beta}}$ es la función logística, y maximizamos el logaritmo de la función de probabilidad, generando el problema de optimización:

$$\text{Minimizar} \sum_{i=1}^{m}(y_i log(g_\beta(x_i)) + (1 - y_i)log(1 - g_\beta(x_i))) \tag{4.4}$$

Podemos simplificar esta expresión:

- $y_i log(g_\beta(x_i)) = y_i log(\frac{1}{1+e^{-x_i^T\beta}}) = -y_i log(1 + e^{-x_i^T\beta})$
- $(1 - y_i)log(1 - g_\beta(x_i)) = (1 - y_i)log(1 - \frac{1}{1+e^{-x_i^T\beta}}) = (1 - y_i)log(\frac{1+e^{-x_i^T\beta}-1}{1+e^{-x_i^T\beta}}) =$
  $= (1-y_i)x_i^T\beta-(1-y_i)log(1+e^{-x_i^T\beta}) = (1-y_i)x_i^T\beta-log(1+e^{-x_i^T\beta})+y_i log(1+e^{-x_i^T\beta})$

Finalmente nos queda:

$$\text{Minimizar} \sum_{i=1}^{m}(1 - y_i)x_i^T\beta - log(1 + e^{-x_i^T\beta}) \tag{4.5}$$

Lo natural sería escribir directamente en la sintaxis convencional de R la función log (1+ exp (X %* %beta)). Pero esta representación no es DCP válida, viola la regla de composición C4. La función log(x) es cóncava no decreciente, para verificar la regla el argumento debería ser cóncavo y, sin embargo, exp(x) es convexa. Por tanto, aunque el objetivo sea convexo, CVXR rechazará el problema. Se necesita encontrar otra escritura del problema. CVXR proporciona la función logistic como camino directo para expresar $f(z) = log(1 + e^z)$. Entonces nos quedaría $\sum_{i=1}^{m}(1 - y_i)x_i^T\beta - logistic(-x_i^T\beta)$.

Entonces nuestro problema completo es expresado en CVXR como:

```
library(CVXR)

# Crear una muestra con 20 variables y 1000 observaciones.

n <- 20
m <- 1000
sigma <- 45
DENSITY <- 0.2

set.seed(183991)

# Se genera una muestra normal de tamaño n
beta_true <- stats::rnorm(n)

# Se toma una muestra de tamaño (1-DENSITY)*n de numeros entre 1 y n
# sin repetición
idxs <- sample(n, size = floor((1-DENSITY)*n), replace = FALSE)
```

```
# Hace 0 el valor las componentes dados por idxs
beta_true[idxs] <- 0

# Se crea una matriz m*n cuyas entradas vienen dadas por una muestra
# normal
X <- matrix(stats::rnorm(m*n, 0, 5), nrow = m, ncol = n)

# Genera los datos y: y=x*beta+eps con eps~N(0,sigma)
y <- sign(X %*% beta_true + stats::rnorm(m, 0, sigma))

# Definir el problema
beta <- Variable(n)

obj <- -sum(X[y <= 0, ] %*% beta) - sum(logistic(-X %*% beta))
prob <- Problem(Maximize(obj))
```

Podemos observar que para la función objetivo se ha empleado la función logistic, como se ha explicado anteriormente, pero además no se ha escrito de forma directa $(1-y_i)x_i^T\beta$. Hemos definido y a partir de la función signo, luego toma valores en {-1,0,1}. Necesitamos que y sea binaria y que tome únicamente valores en {0,1}, para ello cuando $y_i = -1$ vamos a considerar que vale cero. La expresión $(1-y_i)x_i^T\beta$ es 0 si $y_i = 1$ y es $x_i^T\beta$ para $y_i = 0$, luego lo que hacemos es tomar las x_i tal que $y_i \leq 0$, es decir, tal que $y_i = 0$ o $y_i = -1$.

```
# Comprobar que el problema es convexo

is_dcp(prob)
```

```
## [1] TRUE
```

```
#Resolver

result <- solve(prob)
beta_log <- result$getValue(beta)

#Podemos ver que solver ha empleado en este caso
result$solver
```

```
## [1] "ECOS"
```

Ahora vamos a resolver el problema usando gml para comparar los resultados:

```
d <- data.frame(y = as.numeric(y > 0), X = X)
glm <- stats::glm(formula = y ~ 0 + X, family = "binomial", data = d)
```

En el cuadro 4.9 tenemos los valores de la solución obtenida por CVXR y glm.

Cuadro 4.9: Resultados del problema Logístico mediante CVXR y glm

	CVXR.est	GLM.est
beta_1	-0.0003345	-0.0003345
beta_2	-0.0149849	-0.0149849
beta_3	0.0144564	0.0144564
beta_4	0.0172645	0.0172645
beta_5	0.0206029	0.0206029
beta_6	-0.0200824	-0.0200824
beta_7	-0.0137066	-0.0137066
beta_8	-0.0009806	-0.0009806
beta_9	-0.0246409	-0.0246409
beta_10	0.0101345	0.0101345
beta_11	-0.0113936	-0.0113936
beta_12	0.0187613	0.0187613
beta_13	0.0051938	0.0051938
beta_14	0.0015573	0.0015573
beta_15	0.0202801	0.0202801
beta_16	-0.0176528	-0.0176528
beta_17	0.0191530	0.0191530
beta_18	0.0131130	0.0131130
beta_19	-0.0265186	-0.0265186
beta_20	0.0039912	0.0039912

Conclusiones

En este trabajo se ha expuesto una metodología para la optimización convexa denominada Programación Convexa Disciplinada. En el capítulo 2 tenemos una larga lista de aplicaciones de la programación convexa, pero existen razones para creer que la cantidad de aplicaciones aumentará drásticamente si se puede hacer más fácil el hecho de crear, analizar y resolver CPs. Una fuente prometedora de nuevas aplicaciones para la programación convexa es la extensión y mejora de las aplicaciones existentes para la programación lineal, y si nos fijamos en su historia vemos que no experimentó un gran desarrollo hasta que se hicieron grandes avances computacionales que facilitaban su resolución.

Por todo esto, la importancia de este método se debe a que en muchos casos verificar si un problema dado es convexo puede ser un trabajo intratable, pero utilizando DCP se puede llevar a cabo de forma automática.

En el capítulo 3 hemos descrito las normas que esta metodología impone sobre la construcción de un problema para así, simplificar el proceso de verificar si es convexo. De forma que la verificación de la convexidad consistirá simplemente en probar que el problema está construido a partir de funciones de la librería de acuerdo a las normas establecidas. Es necesario destacar que el cumplimiento de estas reglas es una condición suficiente de convexidad, no necesaria, es decir, la programación Convexa Disciplinada no se puede utilizar para rechazar que un problema sea convexo. Además, ofrece otras ventajas como el hecho de que, mediante las implementaciones gráficas, permite trabajar con funciones no diferenciables sin la pérdida de rendimiento que normalmente va asociada a ellas, como hemos podido comprobar en el capítulo 3 mediante un ejemplo con la función mínimo. Otra de sus principales ventajas es su generalidad, puede trabajar con cualquier tipo de problema, no se restringe a un tipo determinado de formas estándar como hacen otras metodologías. Esta generalidad no se pierde por el hecho de que existan un conjunto de reglas determinado, ya que hemos visto que la librería en la que se basa es extensible.

En el último capítulo hemos llevado esta metodología a la práctica a través del paquete CVXR de R. Como se ha visto en los ejemplos y aplicaciones planteados, dado un problema, CVXR emplea la programación convexa disciplinada previamente a resolver, para probar que sea convexo. Una vez probada la convexidad, utiliza las implementaciones gráficas para transformar el problema y lo pasa a un solver, como ECOS o SCS, para resolverlo.

Permite formular los problemas de una forma muy natural e intuitiva, utilizando la amplia lista de funciones que tiene implementada. Además, es posible formular y resolver problemas para los que no exista un código personalizado. Como se ha mostrado en la

resolución del problema de mínimos cuadrados, puede no ser el paquete óptimo para resolver ciertos problemas comunes para los cuales ya exista un paquete exclusivo, ya que CVXR probablemente sea más lento que un algoritmo personalizado y serían necesarias más líneas de código, pero estos paquetes están limitados. Si necesitaramos añadir nuevas restricciones CVXR sería el paquete ideal, debido a su flexibilidad. Como hemos visto en las aplicaciones, es muy sencillo incluir nuevas restricciones o transformar un problema añadiendo pocas líneas de código.

Bibliografía

[1] Achtziger, W., Bendsoe, M., Ben-Tal, A. and J Zowe 1992. Equivalent displacement based formulations for maximum strength truss topology design. *Impact of Computing in Science and Engineering,*

[2] Alizadeh, F. 1995. Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM Journal on Optimization.*

[3] Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W. and Iannone, R. 2018. *Rmarkdown: Dynamic documents for r.*

[4] Andersen, E. and Andersen, K. 2000. The mosek interior point optimizer for linear programming: An implementation of the homogeneous algorithm. *In High Performance Optimization.*

[5] Avriel, M., Dembo, R. and Passy, U. 1975. Solution of generalized geometric programs. *International Journal for Numerical Methods in Engineering.*

[6] Ben-Tal, A. and Bendsoe, M. 1993. A new method for optimal truss topology design. *SIAM J. Optim.*

[7] Bendsoe, M., A Ben-Tal and Zowe, J. 1994. Optimization methods for truss geometry and topology design. *Structural Optimization.*

[8] Benson, S. 2002. *DSDP 4.5: A daul scaling algorithm for semidefinite programming.* Web site: http://www-mix.mcs.an1.gov/benson/dsdp/.

[9] Bertsimas, D. and Nino-Mora, J. 1999. Optimization of multiclass queuing networks with changeover times via the achievable region approach: Part ii, the multistation case. *Mathematics of Operations Research.*

[10] Biswas, P. and Ye, Y. 2004. Semidefinite programming for ad hoc wireless sensor network localization. *Technical report, Stanford University.*

[11] Borchers, B. 1999. CDSP, a c library for semidefinite programming. *Optimization Methods and Software.*

[12] Boyd, S. and Vandenberghe, L. 2004. Convex optimization. *Cambridge University Press.*

[13] Brooke, A., Kendrick, D., Meeraus, A. and Raman, R. 1998. *GAMS: A user's guide.* The Scientific Press. Web site: http: //www . gams . com/docs/gams/GAMSUsersGuide.pdf.

[14] Calafiore, G. and Indri, M. 2000. Robust calibration and control of robotic manipulator. *In American Control Conference.*

[15] Chinneck, J. 2003. *MProbe 5.0 (software package).* Web site: http://www.sce.carleton.ca/

faculty/chinneck/mprobe.html.

[16] Cox, D.R. 1958. The regression analysis of binary sequences. *Journal of the Royal Statistical Society.*

[17] Crusius, C. 2002. A parser/solver for convex optimization problems. *PhD thesis, Stanford University.*

[18] Dantzig, G.B. 1963. Linear programming and extensions. *Princeton University Press.*

[19] Davidson, T., Luo, Z. and Wong, K. 2000. Design of orthogonal pulse shapes for communications via semidefinite programming. *IEEE Transactions on Communications,*

[20] Diamond, S. and Boyd, S. 2016. CVXPY: A python-embedded modeling language for convex optimization. *Journal of Machine Learning Research.*

[21] Doherty, A., Parrilo, P. and Spedalieri, F. 2002. Distinguishing separable and entangled states. *Physical Review Letters.*

[22] Domahidi, A., Chu, E. and Boyd, S. 2013. ECOS: An socp solver for embedded systems. *In Proceedings of the European Control Conference.*

[23] Dufin, R. 1970. Linearizing geometric programs. *SIAM Review.*

[24] Fourer, R., Gay, D. and Kernighan, B. 1999. AMPL: A modeling language for mathematical programming. *Duxbury Press.*

[25] Freedman, D.A. 2009. Statistical models: Theory and practice. *Cambridge University Press.*

[26] Fu, Anqi, Narasimhan, B. and Boyd, S. 2017. CVXR: An r package for disciplined convex optimization.

[27] Fujisawa, K., Kojima, M., Nakata, K. and Yamashita, M. 2002. SDPA (semi-definite programming algorithm) user's manual-version 6.00. *Technical report, Tokyo Insitute of Technology.*

[28] Gelfand, I.M. and Fomin, S.V. 1963. Calculus of variations. *Prentice-Hall.*

[29] GenBeta https://www.genbeta.com/desarrollo/r-un-lenguaje-y-entorno-de-programacion-para-analisis-estadistico.

[30] Ghaoui, L.E., Commeau, J.-L., Delebecque, F. and Nikoukhah, R. 1999. *LMITOOL 2.1 (software package).".* Web site: http://robotics.eecs.berkeley.edu/elghaoui/lmitool/lmitool.html.

[31] Giiler, 0. and Hauser, R. 2002. Self-scaled barrier functions on symmetric cones and their classification. *Foundations of Computational Mathematics.*

[32] Goldfarb, D. and Iyengar, G. 2002. Robust portfolio selection problems. *Technical report, Computational Optimization Research Center, Columbia University.*

[33] Grant, M. 2004. Disciplined convex programming. *PhD thesis, Department of Electrical Engineering, Stanford University.*

[34] Grant, M. and Boyd, S. 2014. CVX: MATLAB software for disciplined convex

programming, version 2.1.

[35] Grant, M., Boyd, S. and Ye, Y. 2006. Disciplined convex programming. *Global Optimization: From Theory to Implementation, Nonconvex Optimization and its Applications*.

[36] Griva, I.A. and Vanderbei, R.J. 2005. Case studies in optimization: Catenary problem. *Optimization and Engineering*.

[37] Gurobi Optimization, I. 2016. *Gurobi optimizer reference manual.* Disponible en http://www.gurobi.com.

[38] Hershenson, M., Mohan, S., Boyd, S. and Lee, T. 1999. Optimization of inductor circuits via geometric programming. *In Proceedings 36th IEEE/ACM Integrated Circuit Design Automation Conference*.

[39] Huaizhong, L. and Fu, M. 1997. A linear matrix inequality approach to robust h, filtering. *IEEE Transactions on Signal Processing*.

[40] Karmarkar, N. 1984. A new polynomial-time algorithm for linear programming. *Combinatorica*.

[41] Kortanek, K., Xu, X. and Ye, Y. 1997. An infeasible interior-point algorithm for solving primal and dual geometric progams. *Mathematical Programming*.

[42] Lasserre, J. 2002. Bounds on measures satisfying moment conditions. *Annals of Applied Probability*.

[43] Lasserre, J. 2001. Global optimization with polynomials and the problem of moments. *SIAM Journal of Optimization*.

[44] Lenstra, J.K., Rinnooy-Kan, A.H.G. and Schrijver, A. 1991. *History of mathematical programming: A collection of personal reminiscenses*. Elsevier Science Ltd.

[45] Lobo, M., Vandenberghe, L., Boyd, S. and Lebret, H. 1998. Applications of second-order cone programming. *Linear Algebra and its Applications*.

[46] Lofberg, J. 2004. YALMIP: A toolbox for modeling and optimization in matlab. *In Proceedings of the IEEE International Symposium on Computed Aided Control Systems Design*.

[47] Lovasz, L. 1986. An algorithmic theory of numbers, graphs and convexity. *Regional Conference Series in Applied Mathematics. SIAM, Philadelphia*.

[48] Luque-Calvo, P.L. 2017. *Escribir un trabajo fin de estudios con r markdown.* Disponible en http://destio.us.es/calvo.

[49] Miller, J., Quigley, P. and Zhu, J. 2015. *CVXcanon: Automatic canonicalization of disciplined convex programs.* https://stanford.edu/class/ee364b/projects/2015projects/reports/miller_quigley_zhu_report.pdf.

[50] MOSEK-ApS 2001. *Mosek (software package).* Web site: http://www.mosek.com.

[51] Nesterov, Y. and Nemirovsky, A. 1988. A general approach to polynomial-time algorithms design for convex programming. *Technical report, Centr. Econ. & Math. Inst., USSR Acad. Sci., Moscow, USSR*.

[52] Nesterov, Y. and Nemirovsky, A. 1993. Interior-point polynomial algorithms in convex programming: Theory and algorithms. *Society of Industrial and Applied Mathematics*

(SIAM) Publications.

[53] Nesterov, Y. and Todd, M. 1997. Self-scaled barriers and interior-point methods for convex programming. *Mathematics of Operations Research.*

[54] Orban, D. and Fourer, R. 2004. DrAmpl: A meta-solver for optimization. *Technical report, Ecole Poytechnique de Montreal.*

[55] O'Donoghue, B., Chu, E., Parikh, N. and Boyd, S. 2016. Conic optimization via operator splitting and homogeneous self-dual embedding. *Journal of Optimization Theory and Applications.*

[56] Park, J., Cho, H. and Park, D. 1999. Design of gbsb neural associative memories using semidefinite programming. *IEEE Transactions on Neural Networks.*

[57] Peton, 0. and Vial, J.-P. 2001. *A tutorial on accpm: User's guide for version 2.01.* Technical Report 2000.5, HEC/Logilab, University of Geneva. http://ecolu-info.unige.ch/-logilab/software/accpm/ accpm. html/.

[58] R Core Team 2016. *R: A language and environment for statistical computing.* R Foundation for Statistical Computing.

[59] Rasmussen, L., Lim, T. and Johansson, A. 2000. A matrix-algebraic approach to successive interference cancellation in cdma. *IEEE Transactions on Communications.*

[60] Renegar, J. 1988. A polynomial-time algorithm, based on newton's method, for linear programming. *Mathematical Programming.*

[61] Rijckaert, M. and Martens, X. 1974. Analysis and optimization of the williams-otto process by geometric programming. *AIChE Journal.*

[62] Rockafellar, R. 1970. Convex analysis. *Princeton Univ. Press, Princeton, New Jersey, second edition.*

[63] RStudio Team 2015. *RStudio: Integrated development environment for r.* RStudio, Inc.

[64] Salazar-Gonzalez, J.J. 2001. *Programación matemática.* Díaz de Santos.

[65] Sturm, J. 1999. Using sedumi 1.02, a matlab toolbox for optimization over symmetric cones. *Optimization Methods and Software.*

[66] Suykens, J.A.K., Gestel, T.V., Brabanter, J.D., Moor, B.D. and walle, J.V. 2002. Least squares support vector machines.

[67] Techopedia "Definition - what does business intelligence (bi) mean?" Disponible en https://www.techopedia.com/definition/345/business-intelligence-bi.

[68] Terlaky, T., Roos, C. and Vial, J.-P. 1997. Interior point approach to linear optimization: Theory and algorithms. *John Wiley & Sons.*

[69] Tutuncu, R., Toh, K. and Todd, M. 2001. SDPT3-a matlab software package for semidefinite-quadratic-linear programming, version 3.0. *Technical report, Carnegie Mello University.*

[70] Udell, M., Mohan, K., Zeng, D., Hong, J., Diamond, S. and Boyd, S. 2014. Convex optimization in julia. *In Proceedings of the Workshop for High Performance Technical*

Computing in Dynamic Languages.

[71] Vandenberghe, L. and Boyd, S. 1996. Semidefinite programming. *SIA M Review*.

[72] Vandenberghe, L., Boyd, S. and Gamal, A.E. 1998. Optimizing dominant time constant in rc circuits. *IEEE Transactions on Computer-Aided Design*.

[73] Vanderbei, R. 2000. LOQO user's manual-version 4.05. *Technical report, Operations Research and Financial Engineering, Princeton University*.

[74] Wickham, H. 2018. *Stringr: Simple, consistent wrappers for common string operations*.

[75] Wickham, H., Chang, W., Henry, L., Pedersen, T.L., Takahashi, K., Wilke, C. and Woo, K. 2018. *Ggplot2: Create elegant data visualisations using the grammar of graphics*.

[76] Wickham, H., François, R., Henry, L. and Müller, K. 2019. *Dplyr: A grammar of data manipulation*.

[77] Wikipedia https://es.wikipedia.org/wiki/Epigrafo.

[78] Wikipedia https://es.wikipedia.org/wiki/Hipografo.

[79] Wikipedia https://es.wikipedia.org/wiki/R_(lenguaje_de_programación).

[80] Williams, H.P. 1978. *Model building in mathematical programming*. John Wiley; Sons Ltd.

[81] Wu, S.-P. and Boyd, S. 2000. SDPSOL: A parser/solver for semidefinite programs with matrix structure. *El Ghaoui and S.-I. Niculescu, editors, Recent Advances in LMI Methods for Control*.

[82] Wu, S.-P., Boyd, S. and Vandenberghe, L. 1998. FIR filter design via spectral factorization and convex optimization. *In B. Datta, editor, Applied and Compu- tational Control, Signals, and Circuits*.

[83] Xie, Y. 2018. *Knitr: A general-purpose package for dynamic report generation in r*.

[84] Ye, Y. 1997. Interior-point algorithms: Theory and practice. *John Wiley & Sons*.

[85] Zhang, S. 2001. A new self-dual embedding method for convex programming. *Technical report, Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong*.

[86] Zibulevsky, M. 1998. Pattern recognition via support vector machine with computationally efficient nonlinear transform. *Technical report, The Univer- sity of New Mexico, Computer Science Department*.