DC programming and DCA combinatorial optimization and polynomial optimization via SDP techniques

Yi Shuai Niu

T H È S E

pour obtenir le titre de

DOCTEUR DE L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE ROUEN

(arrêté ministériel du 7 août 2006)

Spécialité : Mathématiques Appliquées

Présentée et soutenue par

Yi Shuai NIU

— Titre de la thèse —

Programmation DC & DCA en Optimisation Combinatoire et Optimisation Polynomiale via les Techniques de SDP

Codes et Simulations Numériques

soutenue le 28 Mai 2010

Membres du Jury :

Rapporteurs :	
Jean Bernard LASSERRE	Professeur, Directeur de Recherche LAAS-CNRS, Toulouse
Abdel LISSER	Professeur, Université de Paris Sud, Orsay
Examinateurs :	
Frédéric BONNANS	Professeur, Ecole Polytechnique de Paris
Adnan YASSINE	Professeur, Université du Havre
Mohamed DIDI BIHA	Professeur, Université de Caen
Hoai An LE THI	Professeur, Université Paul Verlaine, Metz
Directeur de Thèse :	
Tao PHAM DINH	Professeur, INSA de Rouen

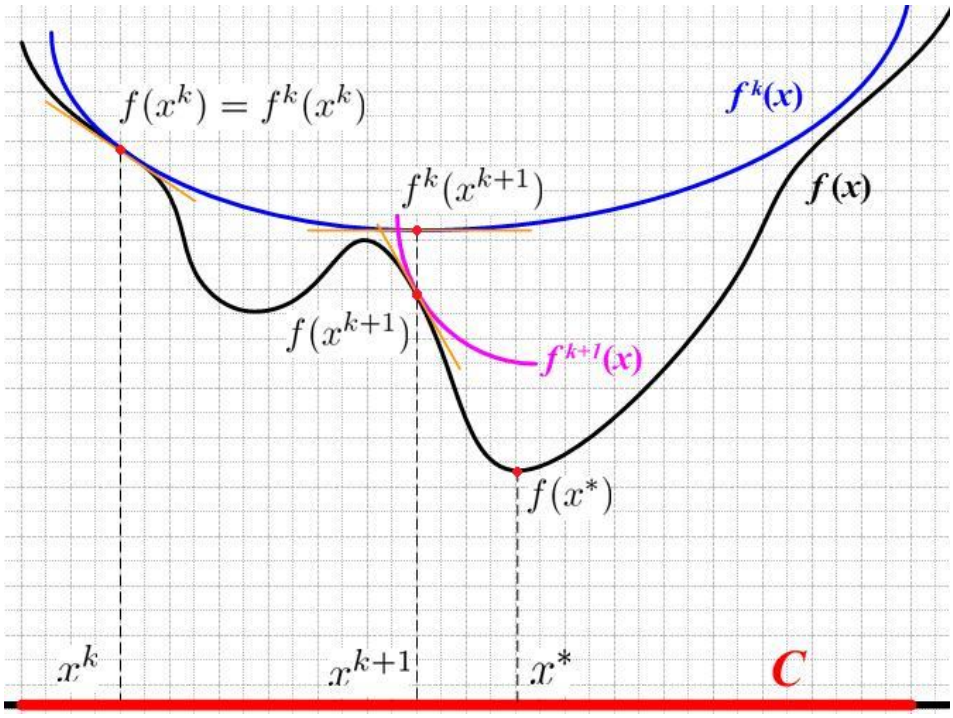
thèse préparée au laboratoire de mathématiques de l'institut national des sciences appliquées de rouen, france

"No great discovery was ever made without a bold guess."

Isaac Newton

"Genius is 1% inspiration and 99% perspiration."

Thomas Edison



Remerciements

Ce travail a été réalisé au sein du Laboratoire de Mathématiques (LMI) de l'Institut National des Sciences Appliquées (INSA) de Rouen, France, sous la direction du Professeur PHAM DINH Tao - Mathématicien de renommée internationale, Pionnier de l'Optimisation DC et Créateur de DCA (DC Algorithm), Directeur de l'Equipe Modélisation et Optimisation du Laboratoire LMI. Qu'il trouve ici ma profonde et sincère gratitude pour son soutien, ses encouragements, et ses conseils scientifiques tout au long de ce travail.

Je suis très heureux et honoré d'exprimer ma profonde et vive reconnaissance aux mathématiciens suivants :

Monsieur le Professeur Witold RESPONDEK - Mathématicien célèbre en Théorie du Contrôle, Directeur du Laboratoire LMI.

Monsieur le Professeur Erik LENGLART - Mathématicien notoire en Probabilité et Calcul Stochastique, Directeur du Département Génie Mathématique, à l'origine de la création du LMI.

Ces deux professeurs m'ont accueilli au sein du Laboratoire LMI, je leur adresse ma profonde gratitude.

Monsieur le Professeur Jean Bernard LASSERRE - Fameux Mathématicien en Optimisation, Récipiendaire du Prix Lagrange, Directeur de Recherche au LAAS - CNRS et Institut de Mathématiques de Toulouse.

Monsieur le Professeur Abdel LISSER - Mathématicien bien connu en Optimisation Combinatoire, Professeur à l'Université de Paris Sud - LRI Orsay, Directeur de l'Equipe Théorie des Graphes et Optimisation Combinatoire de LRI.

Ces deux professeurs m'ont fait l'honneur d'accepter de rapporter cette thèse. Qu'ils reçoivent mes sincères et respectueux remerciements.

Monsieur le Professeur Frédéric BONNANS - Mathématicien célèbre en Optimisation, Professeur à l'Ecole Polytechnique de Paris, CMAP, Directeur de Recherche au INRIA Saclay.

Monsieur le Professeur Adnan YASSINE - Mathématicien réputé en Optimisation, Professeur à l'Université du Havre, Directeur du Laboratoire de Mathématiques Appliquées du Havre (LMAH).

Monsieur le Professeur Mohamed DIDI BIHA - Mathématicien connu en Optimisation Combinatoire, Professeur à l'Université du Havre.

Madame le Professeur LE THI Hoai An - Mathématicienne renommée en Optimisation DC, Professeur à l'Université Paul Verlaine - Metz, Directrice du Laboratoire d'Informatique Théorique et Appliquée (LITA).

Je souhaite leur exprimer toute ma gratitude pour avoir accepté d'être membres du jury.

Je voudrais adresser mes sincères remerciements aux mathématiciens, professeurs, chercheurs : NGUYEN DONG Yen, LE DUNG Muu, Berç RUSTEM, Didier HENRION, LE NGOC Tho, HUYNH VAN Ngai, Michal KOCVARA, DUONG HOANG Tuan, Mohamed YAGOUNI, Nalan GULPINAR ... pour leur soutien, leurs encouragements, et les discussions que nous avons pu avoir.

Je tiens à exprimer mes profondes remerciements à mes chers parents pour m'avoir donné la liberté de suivre mes passions de recherches mathématiques et scientifiques, pour leur soutien éternel, et leurs encouragements permanents pendant ces longues années d'éloignement.

Mes vifs remerciements vont également à Arnaud KNIPPEL, Carole LE GUYADER, Adel HAMDI, NGUYEN CANH Nam, Saul MAPAGHA, PHAM VIET Nga, Mamadou THIAO, Khaled DAHAMNA et tous ceux qui n'ont pas été cités ici, tous les membres du Laboratoire LMI et les membres du Département Génie Mathématique, mes chers collègues, professeurs et mes étudiants de l'INSA, mes amis ... pour leur sincère amitié, leurs services et soutien au cours de ces années.

Table des matières

Introduction

Cette thèse représente une contribution de la Programmation DC (Difference of Convex functions) et DCA (DC Algorithm) à l'optimisation combinatoire et à l'optimisation polynomiale via les techniques de relaxation DC/SDP. La programmation DC et DCA - introduite par Pham Dinh Tao en 1985 et intensivement développée par Le Thi Hoai An et Pham Dinh Tao depuis 1994 ([20]-[42] et http://lita.sciences.univ-metz.fr/~lethi/) - constitue l'épine dorsale de la programmation non convexe et de l'optimisation globale.

Le passage de l'optimisation convexe à l'optimisation non convexe est marqué par l'absence de conditions d'optimalité globale vérifiables, sauf dans des cas rares (par exemple le problème de minimisation d'une forme quadratique sur une boule euclidienne (the trust region subproblem) dans la méthode de région de confiance (Trust Region method) [25] ou celui de minimisation d'un polynôme sous contraintes polynomiales [60], etc). Cela se traduit en pratique, par la quasi-impossibilité de construire des méthodes itératives convergeant vers des solutions globales et, par voie de conséquence, par l'immense difficulté de calculer numériquement une solution globale d'un programme non convexe, surtout en grande dimension.

La programmation DC inclut les problèmes de la programmation convexe et aussi presque tous les problèmes d'optimisation non convexes (différentiables et non différentiables). La forme standard d'un programme DC est

$$(P_{dc}) \qquad \alpha = \inf\{f(x) = g(x) - h(x) : x \in \mathbb{R}^n\}$$

où g, $h : \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$ sont convexes semi-continues inférieurement et propres. Une telle fonction f est appelée fonction DC et g, h sont des composantes DC de f. Toute contrainte représentée par un ensemble convexe fermé $C \subset \mathbb{R}^n$ est prise en compte dans (P_{dc}) par le biais de sa fonction indicatrice χ_C ($\chi_C(x) = 0$ si $x \in C$; $+\infty$ dans le cas contraire) ajoutée à la fonction g.

La dualité DC associe au programme DC primal (P_{dc}) son dual, qui est aussi un programme DC :

$$(D_{dc}) \qquad \alpha = \inf\{h^*(y) - g^*(y) : y \in \mathbb{R}^n\}.$$

Basée sur les conditions d'optimalité locale et sur la dualité en programmation DC, DCA est une méthode de descente sans recherche linéaire. Elle construit deux suites $\{x^k\}$ et $\{y^k\}$ convergeant vers des points de Karush-Kuhn-Tucker (KKT) du problème (P_{dc}) et (D_{dc}) respectivement, telles que les suites $\{g(x^k) - h(x^k)\}$ et $\{h^*(y^k) - g^*(y^k)\}$ convergent vers la même limite.

Pour une utilisation optimale de DCA, il est crucial d'apporter une réponse à ces deux questions :

1. Comment trouver une décomposition DC pour la fonction objectif f bien adaptée à la structure spécifique du problème traité ?
2. Quelle stratégie d'initialisation utiliser pour DCA ?

En pratique, DCA est peu coûteux et ainsi capable de traiter les programmes non convexes de grande dimension.

Les programmes non convexes étudiés dans cette thèse ne sont pas à l'origine des programmes DC : il s'agit plutôt de la minimisation d'une fonction DC sous contraintes DC. Nous les traitons comme des programmes DC pénalisés avec ou sans utilisation des techniques de relaxation DC/SDP. La combinaison de DCA avec les algorithmes de l'optimisation globale dont le plus célèbre "Séparation et Évaluation Progressive" ou simplement "Séparation et Évaluation" (SE), en anglais Branch-and-Bound (B&B) vise à

1. Contrôler le caractère global des solutions calculées par DCA et déterminer le cas échéant une meilleure solution pour relancer DCA.
2. Améliorer les bornes supérieures et accélérer la convergence de B&B afin de pouvoir traiter des programmes DC de plus grande dimension.

Quand aux bornes inférieures, nous faisons appel à des techniques de relaxation DC/SDP de manière appropriée dans la mesure du possible.

Dans cette thèse, nous nous intéressons particulièrement aux trois grandes familles de problèmes d'optimisation non convexes suivantes :

1. La programmation mixte avec variables entières.
2. La programmation avec des fonctions polynomiales.
3. La programmation sous contraintes de matrices semi-définies.

1. La programmation mixte avec variables entières

Cette partie comprend les chapitres 5-6. Le problème d'optimisation mixte avec variables entières peut être défini comme :

$$\min \quad f(x,y) := x^T Q_0 x + y^T P_0 y + c_0^T x + d_0^T y \tag{1}$$

sous contraintes :

$$Ax + By \leq b, A_{eq}x + B_{eq}y = b_{eq}, x \in [\underline{x}, \bar{x}], y \in [\underline{y}, \bar{y}] \tag{2}$$

$$x^T Q_i x + y^T P_i y + c_i^T x + d_i^T y \leq s_i, i = 1, \ldots, L \tag{3}$$

$$x \in \mathbb{R}^n, y \in \mathbb{Z}^m, \tag{4}$$

où x est une variable continue, et y une variable entière. La fonction objectif, donnée par la formule (1), est une fonction quadratique des variables x et y. Les contraintes (2) représentent plusieurs types de contraintes linéaires : les contraintes d'inégalité linéaires, les contraintes d'égalité linéaires, ainsi que les contraintes d'hyper-rectangles. De plus, les contraintes d'inégalité avec des fonctions quadratiques (3) sont aussi prises en compte. Cette formulation contient un grand nombre de problèmes d'optimisation. Par exemple :

1. Si $Q_i = 0, P_i = 0, d_i = 0, B = 0, B_{eq} = 0, i = 0, \ldots, L$, ce problème devient un programme linéaire (LP).
2. Si $P_i = 0, d_i = 0, B = 0, B_{eq} = 0, i = 0, \ldots, L$, c'est un problème de programmation quadratique sous contraintes linéaires et contraintes quadratiques (QCQP) :
   - Si $Q_i, i = 0, \ldots, L$ sont des matrices semi-définies positives alors c'est un programme convexe.
   - Si l'une des matrices Q_i n'est pas une matrice semi-définie positive alors c'est un programme non convexe.
3. Si les paramètres $P_i, d_i, B, B_{eq}, i = 0, \ldots, L$ ne sont pas tous égaux à zéro alors c'est un programme non convexe mixte avec des variables entières (MIP). De plus, si y est une variable binaire alors c'est un programme mixte avec variables binaires (MIP0-1).

Pour le cas linéaire et/ou quadratique convexe, le problème est classique et il existe plusieurs approches de résolution efficientes et célèbres (comme la méthode du simplexe pour le programme linéaire et la méthode du point intérieur pour le programme quadratique convexe). Nos travaux dans cette partie se consacrent à la résolution des problèmes non convexes selon les critères suivants :

- La variable entière y est-elle binaire ?
- La fonction quadratique est-elle convexe ?

Nous avons ainsi les quatre types de problèmes :

1. Minimisation d'une fonction quadratique convexe mixte avec variables binaires (MQP0-1).
2. Minimisation d'une fonction quadratique convexe mixte avec variables entières (MIQP).
3. Minimisation d'une fonction quadratique non convexe mixte avec variables binaires (MNQP0-1).
4. Minimisation d'une fonction quadratique non convexe mixte avec variables entières (MINQP).

Nos contributions portent sur :

- Les reformulations du problème de la programmation mixte sous la forme de la programmation DC.
- La résolution du programme mixte avec variables entières en utilisant DCA.

- La combinaison de DCA avec un schéma SE afin de construire une nouvelle approche d'optimisation globale pour résoudre le problème de programmation mixte avec variables entières.
- Le développement de logiciels pour les algorithmes proposés et les simulations numériques.

2. La programmation avec des fonctions polynomiales

Dans cette partie, nous travaillons sur la programmation non convexe dont la fonction objectif et les contraintes sont formées par des fonctions polynomiales :

$$\min\{f_0(x) : f_i(x) \leq 0, i = 1, \ldots, m, x \in \mathbb{R}^n\}$$

où $f_i, i = 0, \ldots, m$, sont polynomiales à n indéterminées $x_1, \ldots, x_n$. Ce problème est souvent nonlinéaire et non convexe, très difficile à résoudre. La programmation quadratique est en effet un cas particulier de la programmation polynomiale où toutes les fonctions polynomiales sont au plus d'ordre deux. D'abord, nous étudions le cas particulier où f_0 est une fonction polynomiale homogène (ou la somme de fonctions polynomiales homogènes) sous contraintes convexes. Nous appliquons notre approche au problème de la gestion de portefeuille avec moment d'ordre supérieur en optimisation financière. Ensuite, nous présentons brièvement nos travaux préliminaires sur la programmation quadratique sous contraintes quadratiques, puis généralisons cette approche à la programmation polynomiale.
Les résultats dans cette partie concernent :

- Les reformulations de la programmation polynomiale sous forme de programme DC.
- L'application de DCA pour résoudre le programme polynomial.
- La combinaison de DCA avec un schéma SE, ainsi que l'utilisation des techniques de relaxation DC/SDP pour résoudre globalement ce problème.
- Le développement d'outils informatiques pour les algorithmes proposés et les simulations numériques.

3. La programmation sous contraintes de matrices semi-définies

Dans la troisième partie, nous nous intéressons aux problèmes d'optimisation avec contraintes BMI (Bilinear Matrix inequality) ou QMI (Quadratic Matrix inequality) qui sont importants dans le domaine du contrôle optimal. Une contrainte BMI (resp. QMI) est une contrainte d'inégalité matricielle non linéaire qui peut être considérée comme une généralisation de la contrainte d'inégalité bilinéaire (resp. quadratique). La contrainte BMI est définie comme suit :

$$F(x, y) := F_{00} + \sum_{i=1}^{n} x_i F_{i0} + \sum_{j=1}^{m} y_j F_{0j} + \sum_{i=1}^{n} \sum_{j=1}^{m} x_i y_j F_{ij} \preceq 0 \qquad (1)$$

où les matrices $F_{ij}, i = 0, \ldots, n, j = 0, \ldots, m$ sont des matrices symétriques et donc la fonction $F(x, y)$ est une matrice symétrique des variables $x \in \mathbb{R}^n$ et $y \in \mathbb{R}^m$. Si $x = y$ alors le BMI devient QMI. Les problèmes d'optimisation sous contraintes BMI et QMI sont des problèmes d'optimisation non convexes, souvent très difficiles à résoudre, même pour un problème de réalisabilité d'une contrainte BMI, problème dont le caractère NP-difficile a été démontré [88]. Nous étudions le problème de réalisabilité du BMI et QMI via notre approche de la programmation DC, et nous obtenons les résultats suivants :

- Les reformulations d'un programme sous contraintes BMI et QMI en un programme DC.
- L'utilisation de DCA pour résoudre le problème BMI et QMI.
- La combinaison de SE et DCA pour résoudre ce problème.
- Le développement d'outils informatiques pour les algorithmes proposés et les simulations numériques.

L'organisation de la thèse consiste en quatre parties :

1. La première partie est constituée des chapitres 1 à 3. Les fondements de la programmation DC & DCA sont brièvement introduits dans le chapitre 1. La programmation semi-définie (SDP) et les techniques de relaxation semi-définie sont présentées dans le chapitre 2. Les théories et les approches de Séparation et Evaluation (SE) sont discutées dans le chapitre 3.
2. La résolution des problèmes de la programmation mixte en variables entières avec les approches de programmation DC est étudiée dans la deuxième partie. Elle regroupe les chapitres 4 et 5.
3. Les chapitres 6 et 7 qui sont inclus dans la troisième partie explorent des approches de la programmation DC pour la résolution de programmes polynomiaux.
4. On réserve la dernière partie (chapitre 8) à la résolution de programmes d'optimisation sous contraintes de type matrices semi-définies via nos approches de la programmation DC.

Ces travaux ont été implémentés à l'aide de plusieurs langages de programmation : MATLAB, C/C++, E-language etc. Nous présentons en annexe quelques extraits de nos codes (ou pseudo-codes) pour illustrer l'aspect "développement de logiciels" de notre travail.

Première partie

Principales techniques de la programmation DC et de l'optimisation globale

Chapitre 1

Programmation DC & DCA

Sommaire

Ce chapitre est une brève introduction à la programmation DC et DCA. Un programme DC est formulé comme un problème de minimisation d'une fonction DC (Difference of Convex functions). L'ensemble des fonctions convexes est en effet un sous ensemble des fonctions DC, et la programmation convexe est un cas particulier de la programmation DC. Une des propriétés essentielles de cet ensemble est sa stabilité par rapport aux opérations usuelles en optimisation. Pour les méthodes de résolution de la programmation DC, il y a deux familles d'approches complémentaires mais complètement différentes :

1. **Approches globales de la programmation non convexe :** Ces approches s'intéressent uniquement aux algorithmes globaux, c.à.d., aux algorithmes permettant de calculer des solutions optimales globales de programmes non convexes. Les outils algorithmiques associés sont développés dans l'esprit de l'optimisation combinatoire, avec la différence que l'on travaille dans des domaines continus. C'est Hoang Tuy qui a incidemment implusé par son papier fondateur en 1964 [10] la nouvelle optimisation globale relative à la maximisation convexe sur polyèdre convexe. Parmi les contributions les plus importantes à ces approches, il faudrait citer celles de Hoang Tuy, R. Horst, H. Benson, H. Konno, P. Pardalos, Le Dung Muu, Le Thi Hoai, Nguyen Van Thoai, Phan Thien Thach et Pham Dinh Tao [49, 50, 51, 23].

Durant les deux dernières décennies, d'énormes progrès ont été accomplis, particulièrement dans les aspects numériques. On peut maintenant résoudre globalement des programmes non convexes spécialement structurés provenant d'applications de plus grandes dimensions, en particulier des programmes non convexes de grande taille mais de faible rang de non-convexité [11]. Cependant, les algorithmes globaux les plus robustes et les plus performants ne permettent pas d'atteindre le but cherché : résoudre des programmes non convexes concrets avec leurs vraies grandes dimensions. Parallèlement à ces approches combinatoires de l'optimisation continue globale, l'approche convexe de la programmation non convexe, a été moins travaillée.

2. **Approches convexes de la programmation non convexe :** Cette approche est basée sur les outils d'analyse convexe, la dualité DC et l'optimalité locale en programmation DC. La programmation DC (Difference of Convex functions) & DCA (DC Algorithm), qui constituent l'épine dorsale de la programmation non convexe, sont introduits en 1985 par Pham Dinh Tao [15, 16, 17, 18, 19] et intensivement développés par Le Thi Hoai An & Pham Dinh Tao depuis 1994 ([20]-[32]) pour devenir maintenant classiques et de plus en plus utilisés par des chercheurs et praticiens de par le monde, dans différents domaines des sciences appliquées. Leur popularité réside dans leur robustesse et leur performance, comparées à des méthodes existantes, leur adaptation aux structures des problèmes traités et leur capacité à résoudre des problèmes industriels de grande dimension. La programmation DC étudie la structure de $DC(\mathbb{R}^n)$, la dualité DC, les relations entre les programmes DC primal et dual et les conditions d'optimalité locale et globale. La complexité des programmes DC réside évidemment dans la distinction entre les solutions optimales globales et locales et, comme conséquence, l'absence de conditions d'optimalité globale faciles à mettre en pratique. A notre connaissance, DCA est à l'heure actuelle l'unique algorithme effectif voué à l'optimisation non convexe et non différentiable de grande dimension. Remarquons qu'une fonction DC f admet une infinité de décompositions DC dont dépend DCA : il y a autant de DCA que de décompositions DC de f. Et les impacts de ces décompositions DC sur les qualités des DCA correspondants (rapidité, robustesse, globalité, ...) sont importants. La question de décomposition DC optimale pour une fonction DC est largement ouverte. En pratique on se contente de décompositions DC bien adaptées aux programmes DC traités, i.e., des décompositions DC qui rendent le calcul des suites $\{x^k\}$ et $\{y^k\}$ facile et peu coûteux. Notons qu'avec des décompositions appropriées, DCA permet de retrouver la plupart des méthodes standard en programmation convexe.

DCA est une méthode de descente sans recherche linéaire (très avantagé par les grandes dimensions) qui est appliquée avec grand succès à de nombreux et divers problèmes d'optimisation non convexe des domaines des science appliquées : Transport-Logistique, Télécommunication, Génomique, Finance, Data Mining, Cryptologie, Mécanique, Traitement d'Image, Robotique & Vision par Ordinateur, Pétrochimie, Contrôle Optimal et Automatique, Problèmes Inverses et Problèmes Mal Posés, Programmation Multiobjectif, Problèmes d'Inégalités Variationnelles (VIP), ... pour ne citer qu'eux.

En pratique DCA converge presque toujours vers des solutions optimales globales. La globalité de DCA peut être vérifiée soit lorsque les valeurs optimales sont connues a priori, soit grâce à des techniques de l'optimisation globale dont la plus populaire reste celles de séparation et évaluation (B&B). C'est ainsi qu'au au cours de cette dernière décennie des approches combinées de DCA et B&B ont été introduits pour résoudre globalement des programmes DC [21]. A ce propos, la structure DC de la fonction objectif $f = g - h$ se prête bien à la recherche d'une minorante convexe de f sur un convexe fermé borné C pour la procédure d'évaluation dans B&B. En effet, au lieu de calculer l'enveloppe convexe de f - ce qui est très difficile, voire impossible en général - on peut se contenter d'une bonne minorante convexe de f comme la somme de g et de l'enveloppe convexe de $-h$ sur C. Cela est bien accessible, sachant que si C est un polyèdre convexe borné, l'enveloppe convexe d'une fonction concave est une fonction convexe polyèdrale qui se calcule à l'aide d'un programme linéaire. De plus si C est un pavé (box) et h est séparable ou si C est un simplexe, cette enveloppe convexe devient une fonction affine explicite. Nos différents travaux montrent que l'algorithme combiné DCA et B&B permet d'accélérer la convergence de la technique B&B usuelle en améliorant de manière significative les bornes supérieures (on relance DCA à partir d'une solution réalisable courante $\overline{x}$ avec la plus petite valeur objectif (la plus petite borne supérieure courante) chaque fois qu'une valeur de la fonction f en $\overline{x}$ diminue).

Parallèlement, des techniques de points intérieurs, de la programmation semi-définie positive (SDP), de décomposition de Benders, de décomposition de Dantzig-Wolfe, de lift & project, des approches polyédrales etc, et des nouvelles coupes planes, ont été combinées de manière convenable avec DCA afin de résoudre des programmes DC de plus grande dimension. D'un autre côté, l'utilisation récente des fonctions subanalytiques [45] a permis un raffinement de la qualité de convergence de DCA pour certaines classes de programmes DC usuels.

Last but not least, on a été obtenu des nouveaux résultats importants [43, 44] sur la pénalité exacte en programmation DC avec ou sans bornes d'erreur (error bounds) des contraintes C non convexes, plus précisément lorsque C est de la

forme $(k, l \in \Gamma_0(\mathbb{R}^n))$

$$C := \{ x \in \mathbb{R}^n : k(x) - l(x) \leq 0\}.$$

Autrement dit, on est face à un problème de minimisation d'une fonction DC sous une contrainte DC. Cette classe de problèmes d'optimisation non convexe est plus large que celle des programmes DC et a fortiori plus difficile à traiter à cause de la non convexité apparue dans les contraintes. Heureusement les techniques de pénalité en programmation DC permettent de reformuler ces programmes non convexes doublement DC en des programmes DC équivalents.

Mes travaux s'inscrivent dans la lignée des travaux de Pham Dinh Tao et Le Thi Hoai An. Ils sont basés sur la programmation DC & DCA et leur combinaison avec les techniques d'optimisation globale (SE) et les techniques de relaxation DC/ SDP pour étudier plusieurs classes de problèmes importants d'optimisation non convexe et/ou d'optimisation combinatoire qui sont difficiles à la fois en théorie et en pratique.

1.1 Eléments d'analyse convexe

Dans cette section, nous allons rappeler quelques définitions et théorèmes d'analyse convexe qui fondent la base de la programmation DC. Pour plus de détails en analyse convexe, on pourra se référer aux ouvrages de Rockafellar et de Hiriart-Urruty [47, 48].

Soit X l'espace euclidien $\mathbb{R}^n$ muni du produit scalaire usuel $\langle x, y\rangle = \sum_{i=1}^n x_i y_i = x^T y$ et de la norme euclidienne associée $\|x\|_2 = \sqrt{\langle x, x\rangle}$, où x^T est le transposé du vecteur x. Y désignera l'espace dual de X relatif au produit scalaire qui peut être identifié à X. L'analyse convexe moderne permet aux fonctions de prendre les valeurs $\pm\infty$. On notera $\bar{\mathbb{R}} = \mathbb{R} \cup \{\pm\infty\}$ qui est muni d'une structure algébrique déduite de manière naturelle, avec la convention $(+\infty) - (+\infty) = +\infty$.

Définition 1.1 (Ensemble Convexe) *Un ensemble $C \subset X$ est dit convexe lorsque, pour tout x et y de C, le segment $[x, y]$ est tout entier contenu dans C, c.à.d,*

$$\forall \lambda \in [0, 1], \lambda x + (1 - \lambda)y \in C.$$

La figure 1.1 illustre cette notion.

Définition 1.2 (Enveloppe convexe) *Soit $S \subset X$. L'enveloppe convexe de S, notée $co(S)$, est l'intersection de tous les sous-ensembles convexes de X qui contiennent S. Elle est en effet le plus petit ensemble convexe de X qui contient S.*

Sachant que X est un espace vectoriel de dimension finie, on peut en déduire facilement que $co(S)$ est l'ensemble des points $x \in X$ qui peuvent s'écrire sous forme de combinaison convexe finie d'éléments de S, c.à.d.

$$co(S) = \{x = \sum_{i=1}^m \lambda_i x^i : x^i \in S, \lambda_i \geq 0, \forall i = 1, \ldots, m; \sum_{i=1}^m \lambda_i = 1\}.$$

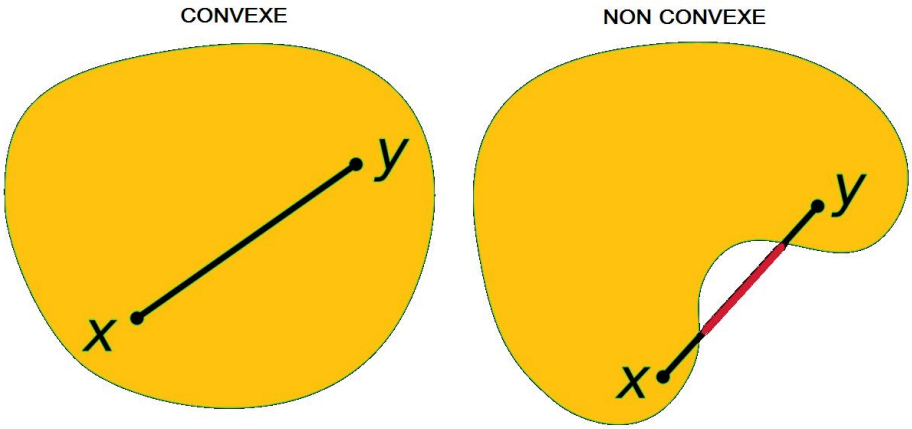


FIG. 1.1 – Ensemble Convexe et Non convexe

Théorème 1.3 (Théorème de Carathéodory) *Dans un espace vectoriel de dimension finie n, $co(S)$ est l'ensemble des barycentres à coefficients positifs ou nuls de familles de $n+1$ points de S.*

Définition 1.4 (Variété affine) *Soit C un ensemble convexe. Une variété affine engendrée par C, notée* $\mathrm{aff}(C)$ *est définie comme :*

$$\mathrm{aff}(C) = \{\sum_i \lambda_i x^i : x^i \in C, \sum_i \lambda_i = 1, \lambda_i \in \mathbb{R}\},$$

où seules les sommes finies sont prises en compte.

Définition 1.5 (Intérieur relatif) *On appelle intérieur relatif d'un ensemble convexe C, noté* $\mathrm{ir}(C)$*, l'intérieur de C relativement à sa variété affine, c.à.d,*

$$\mathrm{ir}(C) = \{x \in C : \exists r > 0, B(x,r) \cap \mathrm{aff}(C) \subset C\},$$

où $B(x,r)$[1] *est la boule euclidienne de centre x et de rayon r.*

En dimension finie, l'intérieur relatif d'un convexe C non vide n'est jamais vide et a la même dimension que C. L'intérieur relatif $ir(C)$ est vide si et seulement si C est aussi vide.

Définition 1.6 (Domaine d'une fonction) *Soit $f : X \to \mathbb{R} \cup \{+\infty\}$. On appelle domaine de f, noté $dom(f)$, l'ensemble*

$$dom(f) = \{x \in X, f(x) < +\infty\}.$$

Définition 1.7 (Fonction propre) *La fonction f est dite propre si elle ne prend jamais la valeur $-\infty$ et si elle n'est pas identiquement égale à $+\infty$.*

[1] $B(x,r) = \{y \in \mathbb{R}^n : \|y - x\| \leq r\}$.

Définition 1.8 (Fonction convexe) *Soit C un ensemble convexe. La fonction $f : C \to \mathbb{R}$ est dite convexe sur C lorsque, pour tout x et y de C et tout $\lambda \in]0,1[$,*

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

Si l'inégalité est stricte pour tout $\lambda \in]0,1[$ et tout x et y de C avec $x \neq y$, alors f est dite strictement convexe.

Cette définition est aussi valable pour $f : C \to \mathbb{R} \cup \{+\infty\}$. Il est clair que si $f : X \to \mathbb{R} \cup \{+\infty\}$ est convexe alors $dom(f)$ est convexe et la restriction $f|_{dom(f)}$ est une fonction convexe à valeurs dans $\mathbb{R}$ sur $dom(f)$.

Remarque 1
- *Soit un ensemble $S \subset X$. On appelle function indicatrice de S, notée $\chi_S(x)$, définie par $\chi_S(x) = 0$ si $x \in S$; et $+\infty$ dans le cas contraire. La fonction $\chi_S(x)$ est convexe si et seulement si S est un ensemble convexe.*
- *Soit $C \subset X$ un ensemble convexe. La fonction $f : C \to \mathbb{R}$ est convexe si et seulement si la fonction $f + \chi_C$ est convexe sur X à valeurs dans $\mathbb{R} \cup \{+\infty\}$.*

Définition 1.9 (Épigraphe) *L'épigraphe de la fonction f noté $\mathrm{epi}(f)$ est l'ensemble*

$$\mathrm{epi}(f) = \{(x, \alpha) \in X \times \mathbb{R} : f(x) \leq \alpha\}.$$

La fonction f est convexe si et seulement si $\mathrm{epi}(f)$ est un sous-ensemble convexe de $X \times \mathbb{R}$.

Définition 1.10 (Fonction fortement convexe) *La fonction $f : C \to \mathbb{R}$ est dite fortement convexe de module ρ sur l'ensemble convexe C (autrement dit ρ-convexe) lorsqu'il existe $\rho > 0$ tel que pour tout x et y de C et tout $\lambda \in [0,1]$, on ait :*

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) - \lambda(1-\lambda)\frac{\rho}{2}\|x-y\|^2.$$

Cela revient à dire que la fonction $f - \frac{\rho}{2}\|.\|^2$ est convexe sur C.
Le module de forte convexité de f sur C, noté $\rho(f, C)$, est défini par :

$$\rho(f, C) = \sup\{\rho > 0 : f - \frac{\rho}{2}\|.\|^2 \text{ est convexe sur C}\}.$$

On dit que f est fortement convexe sur C si $\rho(f, C) > 0$.

Remarque 2 Toute fonction fortement convexe est strictement convexe et toute fonction strictement convexe est convexe.

Définition 1.11 (Sous-gradient) *Soit une fonction convexe propre f sur X. Un vecteur y dans Y est appelé sous-gradient de f au point $x^0 \in \mathrm{dom}(f)$ si pour tout $x \in X$ on a :*

$$f(x) \geq f(x^0) + \langle y, x - x^0 \rangle.$$

Définition 1.12 (Sous-différentiel) *L'ensemble de tous les sous-gradients de f au point $x^0 \in \mathrm{dom}(f)$ est le sous-différentiel de f au point x^0, noté $\partial f(x^0)$.*
Le domaine du sous-différentiel de f est $\mathrm{dom}(\partial f) = \{x : \partial f(x) \neq \emptyset\}$.

Proposition 1.13
- *$\partial f(x)$ est une partie convexe fermée de Y.*
- *$ir(\mathrm{dom}(f)) \subset \mathrm{dom}(\partial f) \subset \mathrm{dom}(f)$.*

Définition 1.14 (ε-sous-gradient) *Soit ε un réel positif. Un vecteur $y \in Y$ est appelé ε-sous-gradient de f au point x^0 si*

$$f(x) \geq f(x^0) + \langle y, x - x^0 \rangle - \varepsilon, \forall x \in X.$$

Le ε-sous-différentiel de f au point x^0, noté $\partial_\varepsilon f(x^0)$, est l'ensemble de tous les ε-sous-gradient de f au point x^0.

Définition 1.15 (Fonction s.c.i.) *La fonction f est dite semi-continue inférieurement (s.c.i.) au point $x \in X$ si*

$$\liminf_{y \to x} f(y) = f(x).$$

On note $\Gamma_0(X)$ l'ensemble des fonctions convexes s.c.i. et propres sur X.

Définition 1.16 (Fonction conjuguée) *La fonction conjuguée de $f \in \Gamma_0(X)$, notée $f^* : Y \to \mathbb{R} \cup \{+\infty\}$, est définie par :*

$$f^*(y) = \sup\{\langle x, y \rangle - f(x) : x \in X\}.$$

Proposition 1.17
- *$f \in \Gamma_0(X) \Longleftrightarrow f^* \in \Gamma_0(Y)$, et $(f^*)^* = f$.*
- *Si $f \in \Gamma_0(X)$ est différentiable en x si et seulement si $\partial f(x)$ se réduit au singleton $\{\nabla f(x)\}$.*
- *Si $f \in \Gamma_0(X)$ alors $y \in \partial f(x) \Longleftrightarrow x \in \partial f^*(y)$.*
- *$y \in \partial f(x) \Longleftrightarrow f(x) + f^*(y) = \langle x, y \rangle$.*
- *$x^0 \in \arg\min\{f(x) : x \in X\} \Longleftrightarrow 0 \in \partial f(x^0)$.*

Définition 1.18 (Polyédre convexe) *Un ensemble convexe C est dit polyédre convexe s'il est l'intersection de nombre fini de demi-espaces de $\mathbb{R}^n$:*

$$C = \bigcap_{i=1}^{m} \{x \in \mathbb{R}^n : \langle a_i, x \rangle - b_i \leq 0, a_i \in \mathbb{R}^n, b_i \in \mathbb{R}\}.$$

.

Définition 1.19 (Fonction polyédrale) *Une fonction $f : \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$ est dite polyédrale si son épigraphe est un ensemble polyédral de $\mathbb{R}^{n+1}$.*

Notons que toute fonction polyédrale est convexe, propre et s.c.i.

Proposition 1.20 *Soit $f \in \Gamma_0(X)$ une fonction convexe. Alors f est polyédrale si et seulement si* $\mathrm{dom}(f)$ *est un ensemble polyédral et*

$$f(x) = sup\{\langle a_i, x\rangle - b_i : i = 1, \ldots, l\} + \chi_{\mathrm{dom}(f)}(x)$$

Proposition 1.21 *Les fonctions convexes polyédrales possèdent les propriétés intéressantes suivantes :*

- *Si f_1 et f_2 sont convexes polyédrales, alors $f_1 + f_2$, $max\{f_1, f_2\}$ sont convexes polyédrales.*
- *Si f est convexe polyédrale alors f^* l'est aussi et* $\mathrm{dom}(\partial f) = \mathrm{dom}(f)$. *De plus, si f est finie partout alors*

$$\mathrm{dom}(f^*) = co\{a_i : i = 1, \ldots, l\},$$

$$f^*(y) = min\{\sum_{i=1}^{l} \lambda_i b_i : y = \sum_{i=1}^{l} \lambda_i a_i, \sum_{i=1}^{l} \lambda_i = 1, \lambda_i \geq 0, i = 1, \ldots, l\}.$$

- *Si f est polyédrale alors $\partial f(x)$ est une partie convexe polyédrale non vide $\forall x \in$* $\mathrm{dom}(f)$.
- *Soient $f_1, \ldots, f_l$ des fonctions convexes polyédrales sur X telles que les ensembles convexes* $\mathrm{dom}(f_i), i = 1, \ldots, l$ *aient un point commun. Alors*

$$\partial(f_1 + \ldots + f_l)(x) = \partial f_1(x) + \ldots + \partial f_l(x), \forall x.$$

1.2 Classe des fonctions DC

Dans cette section, nous allons introduire la définition des fonctions DC et leurs propriétés. Pour plus de détails sur les fonctions DC, on pourra se référer à [12, 13, 51, 21, 29].

Définition 1.22 (Fonction DC) *Soit C un ensemble convexe de X. Une fonction $f : C \to \mathbb{R} \cup \{+\infty\}$ est dite DC sur C si elle peut s'écrire sous la forme*

$$f(x) = g(x) - h(x), \forall x \in C,$$

où g et h sont deux fonctions convexes sur C. On dit que $g - h$ est une décomposition DC de f.

Remarque 3 Si $f = g - h$ est une fonction DC sur C alors pour toute fonction convexe finie p sur C, $f = (g+p) - (h+p)$ est aussi une décomposition DC de f. Ainsi une fonction DC possède une infinité de décompositions DC.

Soit $Conv(C)$ l'ensemble des fonctions convexes sur C. L'ensemble des fonctions DC sur C, noté $DC(C)$, est l'espace vectoriel engendré par $Conv(C)$ défini comme

$$DC(C) = Conv(C) - Conv(C).$$

L'espace vectoriel $DC(C)$ est une classe de fonctions assez large. Il contient toutes les fonctions convexes sur C, les fonctions concaves sur C, ainsi que beaucoup de fonctions ni convexes ni concaves sur C. Il contient la quasi-totalité des fonctions rencontrées dans les problèmes concrets en optimisation non convexe.

Les fonctions DC possèdent beaucoup de propriétés importantes qui ont été établies à partir des années 50 par Alexandroff (1949) [12] et Hartman (1959) [13] etc. La classe de fonctions DC est stable par rapport aux opérations usuelles utilisées en optimisation.

Proposition 1.23 *Si $f = g - h, f_i = g_i - h_i, i = 1, \ldots, m$ sont des fonctions dans $DC(C)$ alors*

- *Une combinaison linéaire des fonctions DC est une fonction DC, c.à.d.,*

$$\sum_{i=1}^{m} \lambda_i f_i \in DC(C), \forall \lambda_i \in \mathbb{R}.$$

- $\max_{i=1,\ldots,m}(f_i) \in DC(C)$ *et* $\min_{i=1,\ldots,m}(f_i) \in DC(C)$ *car*

$$\max_{i=1,\ldots,m} f_i = \max_{i=1,\ldots,m} [g_i + \sum_{j=1, j\neq i}^{m} h_j] - \sum_{j=1}^{m} h_j,$$

$$\min_{i=1,\ldots,m} f_i = \sum_{j=1}^{m} g_j - \max_{i=1,\ldots,m} [h_i + \sum_{j=1, j\neq i}^{m} g_j].$$

- $|f| \in DC(C)$, *car*

$$|f| = 2\max(g, h) - (g + h).$$

- $f^+, f^- \in DC(C)$ *où* $f^+(x) = \max(0, f(x))$ *et* $f^-(x) = \min(0, f(x))$ *car*

$$f^+ = max(g, h) - h$$

$$f^- = g - max(g, h)$$

- $\prod_{i=1}^{m} f_i \in DC(C)$

Définition 1.24 (Fonction localement DC) *Supposons que l'ensemble convexe C est ouvert. Une fonction $f : C \to \mathbb{R} \cup \{+\infty\}$ est dite localement DC sur C si pour tous les points $x^0 \in C$, il existe une boule de centre x^0 et de rayon ε, notée $B(x^0, \varepsilon)$, telle que f est une fonction DC sur $B(x^0, \varepsilon), \forall \varepsilon \geq 0$.*

Soit C un ouvert convexe de X. On note $C^{1,1}(C)$ le sous-espace vectoriel de $C^1(C)$ des fonctions dont le gradient est localement lipschitzien sur C. $C^{1,1}(C)$ est contenu dans le cône convexe des fonctions $LC^2(C)$, dites fonctions de $sous-C^2$, qui sont localement enveloppe supérieure d'une famille de fonctions de classe $C^2(C)$. Le théorème suivant montre la richesse de la classe de fonctions DC :

Théorème 1.25
- *Une fonction $sous-C^2$ sur C est une fonction localement DC sur C.*
- *Une fonction localement DC sur C est une fonction DC sur C.*

Finalement, on a :

$$C^2(C) \cup C^{1,1}(C) \cup C^1(C) \cup Conv(C) \cup LC^2(C) \subset DC(C).$$

Si C est en plus compact, alors DC(C) est un sous-espace vectoriel dense dans l'ensemble des fonctions continues sur C muni de la norme de la convergence uniforme sur C.

1.3 Programmation DC

Un programme DC est défini comme :

$$(P_{dc}) \qquad \inf\{f(x) = g(x) - h(x) : x \in \mathbb{R}^n\},$$

où g et h sont convexes. C'est un programme sans contrainte. En effet, beaucoup de problèmes d'optimisation non convexe sous contraintes (convexes et nonconvexes) peuvent être transformés en (P_{dc}) sous certaines conditions techniques. Les problèmes suivants sont bien connus en optimisation non convexe.

(1) $\sup\{f(x) : x \in C\}$, f et C sont convexes.

(2) $\inf\{f_0(x) = g_0(x) - h_0(x) : x \in C\}$, g_0, h_0 et C convexes.

(3) $\inf\{f_0(x) = g_0(x) - h_0(x) : x \in C, g_i(x) - h_i(x) \leq 0, i = 1, \ldots, m\}$, g_i, h_i, $i = 0, \ldots, m$ et C sont convexes.

Le programme (1) est équivalent à (P_{dc}) dans la mesure où, d'une part, on peut transformer le programme (1) en (P_{dc}) avec $g = \chi_C$ (la fonction indicatrice de C définie par $\chi_C(x) = 0$ si $x \in C$; $+\infty$ dans le cas contraire) et $h = -f$; d'autre part, (P_{dc}) peut se réécrire sous la forme du programme (1) avec une variable additionnelle t telle que

$$\sup\{h(x) - t : g(x) - t \leq 0\}.$$

Selon la même technique, le programme (2) (un programme DC sous contrainte convexe) est aussi équivalent à (P_{dc}) avec $g = g_0 + \chi_C$ et $h = h_0$.

Dans le programme (3), les contraintes $g_i(x) - h_i(x) \leq 0, i = 1, \ldots, m$ sont des contraintes nonconvexes, appelées *contraintes DC*, qui sont équivalentes à une

seule contrainte DC $\hat{g} - \hat{h} \leq 0$ du fait que $\{g_i(x) - h_i(x) \leq 0, i = 1, \ldots, m\} \equiv \{\max_{i=1,\ldots,m}(g_i(x) - h_i(x)) \leq 0\}$ (voir la Proposition 1.23). Le programme suivant est donc un programme équivalent au programme (3)

$$\inf\{f_0(x) = g_0(x) - h_0(x) : x \in C, \hat{g}(x) - \hat{h}(x) \leq 0\}.$$

Grâce au théorème de pénalité exacte relative à la contrainte DC $\hat{g}(x) - \hat{h}(x) \leq 0$, on peut ramener ce dernier programme à la forme (P_{dc}).

Ainsi, la résolution de (P_{dc}) entraîne celle des autres. Le programme (P_{dc}) est par conséquent un problème essentiel en programmation DC. La structure spéciale de (P_{dc}) a permis d'importants développements, tant sur le plan théorique que sur le plan algorithmique.

Sur le plan théorique, on dispose d'une élégante théorie de la dualité et de conditions d'optimalité (voir [52, 46, 21, 18]). Du point de vue algorithmique, un algorithme remarquable "DCA", a été introduit par Pham Dinh Tao [17, 18, 19]) à l'état préliminaire en 1985 et développé ensuite par Pham Dinh Tao, Le Thi Hoai An depuis 1994 ([20]-[42]). Dans la suite de ce chapitre, nous allons présenter la dualité, les conditions d'optimalité et DCA.

1.4 Dualité en programmation DC

Le concept de dualité est une notion fondamentale en mathématiques. En analyse convexe, une théorie de la dualité a été proposée depuis plusieurs décennies [47]. Plus récemment, la dualité a été introduite et développée en analyse non convexe : tout d'abord, dans le cas des programmes Quasi-convex et Anti-convex [54, 55], puis dans le cas des programme DC. La dualité DC introduite par Toland [52] est une généralisation du travail de Pham Dinh Tao sur la maximisation convexe [14]. Pour plus de détails en dualité DC, on pourra se référer au travail de Le Thi Hoai An [21]. Dans cette section, nous présentons les résultats essentiels en dualité DC.

Considérons le programme DC suivant :

$$(P_{dc}) \qquad \alpha = \inf\{f(x) = g(x) - h(x) : x \in X\},$$

où $g, h \in \Gamma_0(X)$. On adopte la convention $(+\infty) - (+\infty) = (+\infty)$ comme en optimisation convexe. Puisque $h \in \Gamma_0(X)$, on a donc $h^{**} = h$ et

$$h(x) = (h^*)^*(x) = \sup\{\langle x, y\rangle - h^*(y) : y \in Y\}.$$

Avec cette relation, on trouve que

$$\begin{aligned}\alpha &= \inf\{g(x) - \sup\{\langle x, y\rangle - h^*(y) : y \in Y\} : x \in X\} \\ &= \inf\{\inf\{g(x) - [\langle x, y\rangle - h^*(y)] : x \in X\} : y \in Y\} \\ &= \inf\{\beta(y) : y \in Y\}\end{aligned}$$

où

$$\begin{aligned}\beta(y) &= \inf\{g(x) - [\langle x, y\rangle - h^*(y)] : x \in X\} \\ &= \begin{cases} h^*(y) - g^*(y), & \text{si } y \in \text{dom}(h^*)\,; \\ +\infty, & \text{sinon.} \end{cases}\end{aligned}$$

Grâce à la convention $(+\infty) - (+\infty) = (+\infty)$, on obtient finalement le problème dual de (P_{dc}), noté (D_{dc}) :

$$(D_{dc}) \qquad \alpha = \inf\{h^*(y) - g^*(y) : y \in Y\}.$$

Le problème (D_{dc}) est aussi un programme DC car h^* et g^* sont deux fonctions convexes dans $\Gamma_0(Y)$. De plus, les (P_{dc}) et (D_{dc}) ont la même valeur optimale.

Il s'avère que la dualité DC est différente de la dualité lagrangienne. "Le théorème de dualité faible" au sens lagrangien n'existe plus dans la dualité DC. En revanche, nous avons le théorème suivant :

Théorème 1.26

- *x^* est une solution optimale globale du problème (P_{dc}) si et seulement si*
$$\alpha = (g - h)(x^*) \leq (h^* - g^*)(y), \forall y \in Y.$$
- *y^* est une solution optimale globale du problème (D_{dc}) si et seulement si*
$$\alpha = (h^* - g^*)(y^*) \leq (g - h)(x), \forall x \in X.$$

Le théorème ci-dessous montre que la résolution du problème primal (P_{dc}) implique la résolution du problème dual (D_{dc}). On observe la parfaite symétrie entre les problèmes primal et dual.

Théorème 1.27 *Soient $g, h \in \Gamma_0(X)$. Alors*

1. $$\inf\{g(x) - h(x) : x \in \text{dom}(g)\} = \inf\{h^*(y) - g^*(y) : y \in \text{dom}(h^*)\}$$
2. *Si y^0 est un minimum de $h^* - g^*$ sur Y alors chaque point $x^0 \in \partial g^*(y^0)$ est un minimum de $g - h$ sur X.*
3. *Si x^0 est un minimum de $g - h$ sur X alors chaque point $y^0 \in \partial h(x^0)$ est un minimum de $h^* - g^*$ sur Y.*

Remarquons qu'il existe des cas particuliers où le problème dual (D_{dc}) est un problème convexe quand son primal (P_{dc}) est non convexe. Par exemple, on peut montrer que :

Théorème 1.28 *Soient $g, h \in \Gamma_0(X)$ et $h(x) = ng(mx)$, alors les fonctions conjuguées h^* et g^* satisfont*

$$h^*(y) = ng^*(\frac{y}{mn}),$$

où $m \neq 0$ et $n > 0$.

Preuve. Par définition de la fonction conjugée, on a

$$h^*(y) = \sup\{\langle x, y\rangle - h(x) : x \in X\} = \sup\{\langle x, y\rangle - ng(mx) : x \in X\}.$$

Sachant que $m \neq 0, n > 0$, alors

$$\sup\{\langle x, y\rangle - ng(mx) : x \in X\} = n \sup\{\langle mx, \frac{y}{mn}\rangle - g(mx) : x \in X\} = ng^*(\frac{y}{mn}).$$

D'où $h^*(y) = ng^*(\frac{y}{mn})$. □

Lemme 1.29 *Soient $g, h \in \Gamma_0(X)$ et $h(x) = ng(\frac{x}{n}), n > 1$. Le problème primal est défini par :*

$$(P) \qquad \inf\{g(x) - h(x) : x \in X\}$$

et son problème dual associé :

$$(D) \qquad \inf\{h^*(y) - g^*(y) = (n-1)g^*(y) : y \in Y\}$$

est un programme convexe.

Preuve. Grâce au Théorème 1.28, on peut supposer que $m = \frac{1}{n}$ et $n > 1$, on a donc $h(x) = ng(\frac{x}{n})$ et $h^*(y) = ng^*(y)$. Dans ce cas, $h^*(y) - g^*(y) = (n-1)g^*(y)$ est une fonction convexe car g^* est convexe et $n - 1 > 0$. □

Voici un exemple concret :

Exemple 1 Soient $x \in \mathbb{R}$, $n > 1$, $g(x) = e^x$ et $h(x) = ne^{\frac{x}{n}}$. Le problème primal

$$(P) \qquad \inf\{g(x) - h(x) = e^x - ne^{\frac{x}{n}} : x \in \mathbb{R}\},$$

est un programme DC non convexe. Son problème dual

$$(D) \qquad \inf\{h^*(y) - g^*(y) = (n-1)g^*(y) : y \in \mathbb{R}\}$$

est un programme convexe.

1.5 Conditions d'optimalité en programmation DC

Soient $\mathcal{P}$ et $\mathcal{D}$ les ensembles de solutions des problèmes (P_{dc}) et (D_{dc}). Un point x^* est dit point critique du problème (P_{dc}) si $\partial g(x^*) \cap \partial h(x^*) \neq \emptyset$ (un point de KKT généralisé).

En optimisation convexe, x^* est un minimum de la fonction convexe f si et seulement si $0 \in \partial f(x^*)$. En programmation DC, la condition d'optimalité est formulée à l'aide des ε-sous-différentiels de g et h.

Théorème 1.30 (Condition d'optimalité globale) *Soient $g, h \in \Gamma_0(X)$ et $f = g - h$. Alors x^* est un minimum global de $g - h$ sur X si et seulement si*

$$\partial_\varepsilon h(x^*) \subset \partial_\varepsilon g(x^*), \forall \varepsilon > 0.$$

Cette caractérisation est une traduction directe du théorème 1.26 et par conséquent invérifiable en pratique.

Remarque 4

1. *Si $f \in \Gamma_0(X)$, on peut écrire $g = f$ et $h = 0$. Dans ce cas, l'optimalité globale de la programmation DC est identique à celle de la programmation convexe : $0 \in \partial f(x^*)$, du fait que $\partial_\varepsilon h(x^*) = \partial h(x^*) = \{0\}, \forall \varepsilon > 0$.*
2. *D'une manière plus générale, considérons les décompositions DC de $f \in \Gamma_0(X)$ de la forme $f = g - h$ avec $g = f + h$ et $h \in \Gamma_0(X)$ finie partout sur X. Le problème DC correspondant est un "faux" problème DC car c'est un problème d'optimisation convexe. Dans ce cas, la condition d'optimalité $0 \in \partial f(x^*)$ équivaut à $\partial h(x^*) \subset \partial g(x^*)$.*

Définition 1.31 *Soient g et h deux fonctions de $\Gamma_0(X)$. Un point $x^* \in dom(g) \cap dom(h)$ est un minimum local de $g - h$ sur X si et seulement si*

$$g(x) - h(x) \geq g(x^*) - h(x^*), \forall x \in V_{x^*},$$

où V_{x^*} désigne un voisinage de x^*.

Théorème 1.32 (Condition nécessaire d'optimalité locale) *Si x^* est un minimum local de $g - h$ alors*

$$\partial h(x^*) \subset \partial g(x^*).$$

Remarque 5 Si h est polyédrale, la condition nécessaire d'optimalité locale est également suffisante.

Corollaire 1.33 *Si $h \in \Gamma_0(X)$ est polyédrale alors une condition nécessaire et suffisante pour que $x^* \in X$ soit un minimum local de $g - h$ est*

$$\partial h(x^*) \subset int(\partial g(x^*)),$$

Théorème 1.34 (Condition suffisante d'optimalité locale) *Si x^* admet un voisinage V tel que*

$$\partial h(x) \subset \partial g(x) \neq \emptyset, \forall x \in V \cap dom(g),$$

alors x^* est un minimum local de $g - h$.

Corollaire 1.35 *Si $x^* \in int(dom(h))$ vérifie*

$$\partial h(x^*) \subset int(\partial g(x^*)),$$

alors x^* est un minimum local de $g - h$.

Théorème 1.36 *1.* $\partial h(x) \subset \partial g(x), \forall x \in \mathcal{P}$ *et* $\partial g^*(y) \subset \partial h^*(y), \forall y \in \mathcal{D}$.

2. ***Transport de minima globaux*** *:*

$$\bigcup_{x \in \mathcal{P}} \partial h(x) \subseteq \mathcal{D} \subset \mathrm{dom}(h^*).$$

La première inclusion devient égalité si g^* *est sous-différentiable dans* $\mathcal{D}$ *(en particulier si* $\mathcal{D} \subset ir(\mathrm{dom}(g^*))$ *ou si* g^* *est sous-différentiable dans* $\mathrm{dom}(h^*)$*) et dans ce dernier cas* $\mathcal{D} \subset (\mathrm{dom}\partial g^* \cap \mathrm{dom}\partial h^*)$.

$$\bigcup_{y \in \mathcal{D}} \partial g^*(y) \subseteq \mathcal{P} \subset \mathrm{dom}(g).$$

La première inclusion devient égalité si h *est sous-différentiable dans* $\mathcal{P}$ *(en particulier si* $\mathcal{P} \subset ir(\mathrm{dom}(h))$ *ou si* h *est sous-différentiable dans* $\mathrm{dom}(g)$*) et dans ce dernier cas* $\mathcal{P} \subset (\mathrm{dom}\partial g \cap \mathrm{dom}\partial h)$.

3. ***Transport de minima locaux*** *: Soit* $x^* \in dom(\partial h)$ *un minimum local de* $g - h$. *Soient* $y^* \in \partial h(x^*)$ *et* V_{x^*} *un voisinage de* x^* *tel que* $g(x) - h(x) \geq g(x^*) - h(x^*), \forall x \in V_{x^*} \cap dom(g)$. *Si*

$$x^* \in int(dom(g^*)) \text{ et } \partial g^*(y^*) \subset V_{x^*},$$

alors y^* *est un minimum local de* $h^* - g^*$.

1.6 DCA (DC Algorithm)

DCA (DC Algorithm) est une méthode itérative d'optimisation locale basée sur l'optimalité locale et la dualité en programmation DC. DCA a été introduit par Pham Dinh Tao [17, 18, 19]) à l'état préliminaire en 1985 et puis intensivement développé par Pham Dinh Tao, Le Thi Hoai An et al. depuis 1994 ([20]-[42]). Cette approche permet de construire deux suites $\{x^k\}$ et $\{y^k\}$ (candidats supposés pour des solutions optimales des programmes DC primal et dual respectivement) telles que leurs limites (x^* et y^*) soient des points KKT généralisés de ces programmes, et telles que les suites $\{(g-h)(x^k)\}$ et $\{(h^*-g^*)(y^k)\}$ soient décroissantes et tendent vers la même limite $\beta = (g-h)(x^*) = (h^*-g^*)(y^*)$. On a les propriétés suivantes :

Proposition 1.37 *1. Les suites* $\{g(x^k) - h(x^k)\}$ *et* $\{h^*(y^k) - g^*(y^k)\}$ *décroissent et tendent vers la même limite* β *qui est supérieure ou égale à la valeur optimale globale* α.

2. Si $(g-h)(x^{k+1}) = (g-h)(x^k)$ *l'algorithme s'arrête à l'itération* $k+1$, *et le point* x^k *(resp.* y^k*) est un point critique de* $g-h$ *(resp.* h^*-g^**).*

3. Si la valeur optimale du problème (P_{dc}) *est finie et si les suites* $\{x^k\}$ *et* $\{y^k\}$ *sont bornées, alors toute valeur d'adhérence* x^* *de la suite* $\{x^k\}$ *(resp.* y^* *de la suite* $\{y^k\}$*) est un point critique de* $g-h$ *(resp.* h^*-g^**).*

Pour construire les deux suites $\{x^k\}$ et $\{y^k\}$, on définit deux programmes convexes (P_k) et (D_k) comme :

$$(D_k) \qquad y^k \in \arg\min\{h^*(y) - [g^*(y^{k-1}) + \langle y - y^{k-1}, x^k\rangle] : y \in Y\}.$$

$$(P_k) \qquad x^{k+1} \in \arg\min\{g(x) - [h(x^k) + \langle x - x^k, y^k\rangle] : x \in X\}.$$

Alors le point x^{k+1} (resp. y^k) est une solution optimale du programme (P_k) (resp. (D_k)). On peut facilement comprendre que (P_k) (resp. (D_k)) est obtenu en remplaçant h (resp. g^*) de (P_{dc}) (resp. (D_{dc})) par sa minorante affine $h_k(x) = h(x^k) + \langle x - x^k, y^k\rangle$ au voisinage de x^k avec $y^k \in \partial h(x^k)$ (resp. $g_k^*(y) = g_k^*(y^{k-1}) + \langle y - y^{k-1}, x^k\rangle$ au voisinage de y^{k-1} avec $x^k \in \partial g^*(y^{k-1})$). Par conséquent, le problème (P_k) (resp. (D_k)) est un problème de la borne supérieure du programme DC (P_{dc}) (resp. (D_{dc})). On a ensuite le schéma simple suivant pour décrire DCA :

$$\begin{array}{ccc} x^k & \longrightarrow & y^k \in \partial h(x^k) \\ & \swarrow & \\ x^{k+1} \in \partial g^*(y^k) & \longrightarrow & y^{k+1} \in \partial h(x^{k+1}). \end{array}$$

DCA est donc un schéma de point fixe $x^{k+1} \in (\partial g^* \circ \partial h)(x^k)$.

Grâce à la proposition 1.37, DCA s'arrête si au moins l'une des suites $\{(g-h)(x^k)\}$, $\{(h^*-g^*)(y^k)\}$, $\{x^k\}$, $\{y^k\}$ converge. En pratique, nous utilisons souvent les conditions d'arrêt suivantes :

- $|(g-h)(x^{k+1}) - (g-h)(x^k)| \leq \varepsilon$.
- $\|x^{k+1} - x^k\| \leq \varepsilon$

pour obtenir une solution ε-optimale. On peut dès lors décrire DCA :

Algorithme DCA

Étape 0 : x^0 donné, $k = 0$

Étape 1 : On calcule $y^k \in \partial h(x^k)$.

Étape 2 : On détermine $x^{k+1} \in \partial g^*(y^k)$.

Étape 3 : Si les conditions d'arrêt sont vérifiées alors on termine DCA ; Sinon $k = k + 1$ et on répète l'Étape 1.

1.6.1 Existence et bornitude des suites générées par DCA

Pour un programme DC, si DCA peut effectivement construire les deux suites $\{x^k\}$ et $\{y^k\}$ à partir d'un point initial arbitraire $x^0 \in X$, alors les deux suites sont dites bien définies.

Lemme 1.38 (Existence des suites) *Les propositions suivantes sont équivalentes :*

1. *Les suites* $\{x^k\}$ *et* $\{y^k\}$ *sont bien définies*
2. $dom(\partial g) \subset dom(\partial h)$ *et* $dom(\partial h^*) \subset dom(\partial g^*)$

La proposition suivante établit les conditions de bornitude pour les suites générées par DCA.

Lemme 1.39 (Bornitude des suites) *Si* $g - h$ *est coercive*[2]*, alors*

1. *la suite* $\{x^k\}$ *est bornée*
2. *si* $\{x^k\} \subset int(dom(h))$ *alors la suite* $\{y^k\}$ *est aussi bornée.*

Si $h^* - g^*$ *est coercive, alors*

1. *la suite* $\{y^k\}$ *est bornée*
2. *si* $\{y^k\} \subset int(dom(g^*))$ *alors la suite* $\{x^k\}$ *est aussi bornée.*

La convergence de DCA est donnée par le théorème suivant :

Théorème 1.40 (Convergence de DCA) *On suppose que les suites* $\{x^k\}$ *et* $\{y^k\}$ *sont bien définies*

1. *Les suites* $\{g(x^k) - h(x^k)\}$ *et* $\{h^*(y^k) - g^*(y^k)\}$ *sont décroissantes et*
   - $g(x^{k+1}) - h(x^{k+1}) = g(x^k) - h(x^k)$ *si et seulement si* $y^k \in \partial g(x^k) \cap \partial h(x^k)$, $y^k \in \partial g(x^{k+1}) \cap \partial h(x^{k+1})$ *et* $[\rho(g) + \rho(h)]\|x^{k+1} - x^k\| = 0$. *De plus, si* g *et* h *sont strictement convexes sur* X*, alors* $x^k = x^{k+1}$*. Dans ce cas, DCA termine en un nombre fini d'itérations.* x^k *et* x^{k+1} *sont des points critiques de la fonction* $g - h$*.*
     ($\rho(g) = \rho(g, X)$ *est le module de forte convexité de* g *sur* X*).*
   - $h^*(y^{k+1}) - g^*(y^{k+1}) = h^*(y^k) - g^*(y^k)$ *si et seulement si* $x^{k+1} \in \partial g^*(y^k) \cap \partial h^*(y^k)$, $x^{k+1} \in \partial g^*(y^{k+1}) \cap \partial h^*(y^{k+1})$ *et* $[\rho(g^*, D) + \rho(h^*, D)]\|y^{k+1} - y^k\| = 0$. *De plus, si* g^* *et* h^* *sont strictement convexes sur* Y*, alors* $y^k = y^{k+1}$*. Dans ce cas, DCA termine en un nombre fini d'itérations.* y^k *et* y^{k+1} *sont des points critiques de la fonction* $h^* - g^*$*.*
2. *Si* $\rho(g, C) + \rho(h, C) > 0$ *(resp.* $\rho(g^*, D) + \rho(h^*, D) > 0$*), alors la suite* $\{\|x^{k+1} - x^k\|^2\}$ *(resp.* $\{\|y^{k+1} - y^k\|^2\}$*) converge.*
3. *Si la valeur optimale du problème* P_{dc} *est finie et si les suites* $\{x^k\}$ *et* $\{y^k\}$ *sont bornées, alors toute valeur d'adhérence* x^* *(resp.* y^**) de* $\{x^k\}$ *(resp.* $\{y^k\}$*) est un point critique de* $g - h$ *(resp.* $h^* - g^*$*).*

[2]une fonction f est coercive si $\lim_{x\to\infty} f(x) = +\infty$.

1.6.2 Programmation DC polyédrale

Dans cette partie, on présente une classe de problèmes DC qui se rencontre fréquemment dans la pratique et possède des propriétés importantes, tant sur le plan théorique que sur le plan algorithmique. Cette classe de problèmes DC s'appelle *DC polyédrale*.

Définition 1.41 (Programmation DC polyédrale) *Un programme DC est dit polyédrale si l'une des composantes convexes (g et h) de la fonction $f = g - h$ est une fonction convexe polyédrale.*

On peut montrer que, pour résoudre un programme DC polyédrale, DCA a une convergence finie.

Notons que h_k est la minorante affine de la fonction convexe h au voisinage de x^k. On construit h^k, fonction convexe polyédrale engendrée par $\{h_k\}$.

$$h_k(x) = h(x^k) + \langle x - x^k, y^k\rangle = \langle x, y^k\rangle - h^*(y^k), \forall x \in X$$

$$h^k(x) = \sup\{h_i(x) : i = 0, \ldots, k\} = \sup\{\langle x, y^i\rangle - h^*(y^i) : i = 0, \ldots, k\}, \forall x \in X,$$

où les suites $\{x^k\}$ et $\{y^k\}$ sont obtenues via la procédure DCA.
On définit de manière analogue la fonction duale g_k^* (resp. $(g^*)^k$)

$$g_k^*(y) = g^*(y^{k-1}) + \langle y - y^{k-1}, x^k\rangle = \langle y, x^k\rangle - g(x^k), \forall y \in Y$$

$$(g^*)^k(y) = \sup\{g_i^*(y) : i = 0, \ldots, k\} = \sup\{\langle y, x^i\rangle - g(x^i) : i = 0, \ldots, k\}, \forall y \in Y.$$

Sachant qu'une fonction convexe propre s.c.i. est caractérisée comme le suprémum de ses minorantes affines, il s'avère donc plus judicieux d'utiliser h^k (resp. $(g^*)^k$) comme sous estimée de la fonction convexe h (resp. g^*) sur X (resp. Y), plutôt que la minorante affine h_k (resp. $(g^*)_k$). On peut ensuite obtenir les programmes suivants :

$$(P^k) \qquad \inf\{g(x) - h^k(x) : x \in X\}$$

$$(D^k) \qquad \inf\{h^*(y) - (g^*)^k(y) : y \in Y\}$$

Les deux problèmes (P^k) et (D^k) sont des problèmes DC non convexes. On peut voir la différence avec les sous-problèmes (P_k) et (D_k) qui sont convexes.

On a le théorème suivant :

Théorème 1.42 *1. $g(x^{k+1}) - h(x^{k+1}) = h^*(y^k) - g^*(y^k)$ si et seulement si $h^k(x^{k+1}) = h(x^{k+1})$.*

2. $h^*(y^k) - g^*(y^k) = g(x^k) - h(x^k)$ si et seulement si $g^*(y^k) = (g^*)^k(y^k)$.

3. $g(x^{k+1}) - h(x^{k+1}) = g(x^k) - h(x^k)$ si et seulement si $h^k(x^{k+1}) = h(x^{k+1})$ et $g^*(y^k) = (g^*)^k(y^k)$.

Par conséquent, si $g(x^{k+1}) - h(x^{k+1}) = g(x^k) - h(x^k) = h^*(y^k) - g^*(y^k)$ alors les propositions suivantes sont vraies :

1. *x^{k+1} (resp. y^k) est une solution optimale du problème (P^k) (resp. (D^k)).*
2. *Si h et h^k coïncident en une solution de (P_{dc}) ou/et g^* et $(g^*)^k$ coïncident en une solution de (D_{dc}), alors x^{k+1} (resp. y^k) est également une solution de (P_{dc}) (resp. (D_{dc})).*

Supposons que la valeur optimale du problème (P_{dc}) est finie et que la suite $\{x^k\}$ générée par DCA est bornée, alors pour toute valeur d'adhérence x^∞ on a

$$g(x^\infty) - h_\infty(x^\infty) = \inf\{g(x^{i+1}) - h_i(x^{i+1}), i = 0, \ldots, \infty\}$$

où h_∞ est la minorante affine de h au voisinage du point x^∞ définie par :

$$h_\infty = h(x^\infty) + \langle x - x^\infty, y^\infty\rangle = \langle x, y^\infty\rangle - h^*(y^\infty), \forall x \in X,$$

avec $y^\infty \in \partial h(x^\infty)$ une valeur d'adhérence de $\{y^k\}$. Le point x^∞ est donc une solution du programme DC

$$(P^\infty) \qquad \inf\{g(x) - h^\infty(x) : x \in X\}$$

De manière analogue, le point y^∞ est une solution du programme DC

$$(D^\infty) \qquad \inf\{h^*(y) - (g^*)^\infty(y) : y \in Y\}$$

On a le théorème suivant :

Théorème 1.43 *Si la valeur optimale du problème (P_{dc}) (resp. D_{dc}) est finie et la suite $\{x^k\}$(resp. $\{y^k\}$) est bornée, alors toute valeur d'adhérence x^∞ (resp. y^∞) est une solution du problème (P^∞) (resp. (D^∞)). De plus, les valeurs optimales sont égales, c.à.d.,*

$$g(x^\infty) - h^\infty(x^\infty) = h^*(y^\infty) - (g^*)^\infty(y^\infty).$$

Si l'une des conditions suivantes est vérifiée :

- *les fonctions h et h^∞ coïncident en une solution optimale de (P_{dc})*
- *les fonctions g^* et $(g^*)^\infty$ coïncident en une solution optimale de (D_{dc}),*

alors x^∞ et y^∞ sont également des solutions optimales de (P_{dc}) et (D_{dc}) respectivement.

1.6.3 Interprétation géométrique de DCA

Notons que DCA ne fonctionne qu'avec les composantes DC g et h. A la k-ième itération de DCA, on remplace la composante h par sa minorante affine $h_k(x) = h(x^k)+\langle x-x^k, y^k\rangle$ au voisinage de x^k. Sachant que h est une fonction convexe, on a donc $h(x) \geq h_k(x), \forall x \in X$. Par suite, $g(x) - [h(x^k) + \langle x - x^k, y^k\rangle] \geq g(x) - h(x), \forall x \in X$. C'est à dire, $g(x) - [h(x^k) + \langle x - x^k, y^k\rangle]$ est une fonction majorante de la fonction $f(x)$. On peut vérifier les propositions suivantes :

Proposition 1.44 *Soit g, h deux fonctions convexes et différentiables sur X. Notons $f^k(x) := g(x) - [h(x^k) + \langle x - x^k, y^k\rangle]$. On a*

- $f^k(x) \geq f(x), \forall x \in X$.
- $f^k(x^k) = f(x^k)$.
- $\nabla f^k(x^k) = \nabla f(x^k)$.

Preuve. Sachant que h est convexe, $h(x) \geq h_k(x), \forall x \in X$. Par conséquent, $f^k(x) = g(x) - [h(x^k) + \langle x - x^k, y^k \rangle] \geq g(x) - h(x) = f(x), \forall x \in X$, c.à.d. $f^k(x) \geq f(x), \forall x \in X$.

$$f^k(x^k) = g(x^k) - [h(x^k) + \langle x^k - x^k, y^k \rangle] = g(x^k) - h(x^k) + 0 = f(x^k).$$

$\nabla f^k(x) = \nabla g(x) - y^k$. Comme g est différentiable, on a $y^k = \nabla h(x^k)$. C'est pourquoi $\nabla f^k(x) = \nabla g(x) - \nabla h(x^k)$. Finalement, on a $\nabla f^k(x^k) = \nabla g(x^k) - \nabla h(x^k) = \nabla f(x^k)$. □

A l'aide de cette proposition, on peut obtenir une simple interprétation géométrique de DCA. En effet, la surface de f^k peut être imaginée comme un "bol" se plaçant au-dessus de la surface de f ; de plus, les deux surfaces se touchent au point $(x^k, f(x^k))$ (voir la figure 1.2).

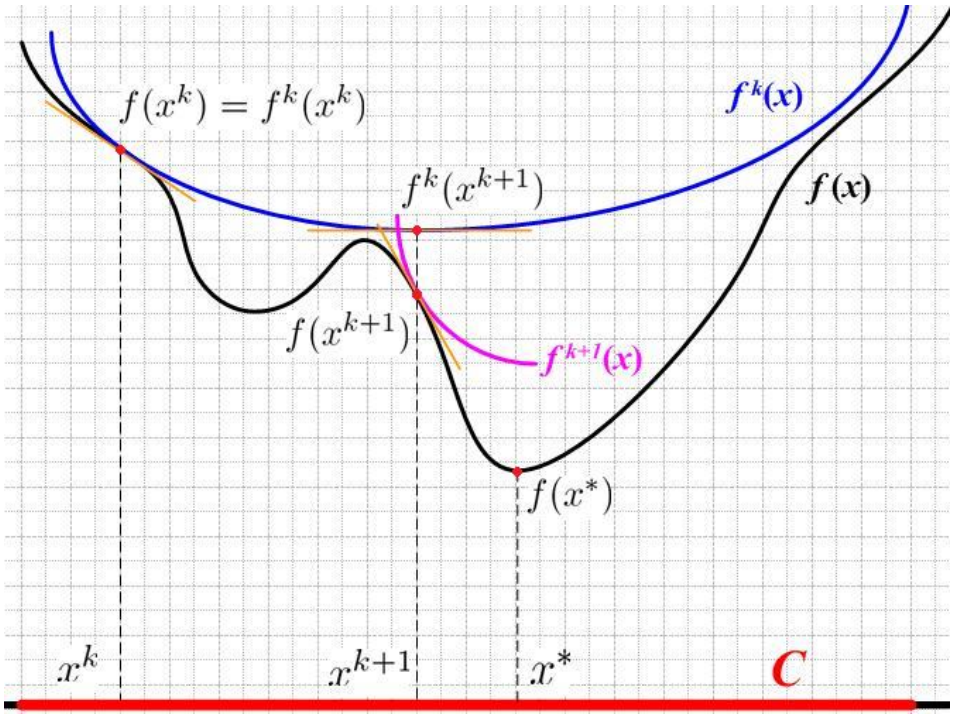


FIG. 1.2 – Interprétation géométrique de DCA

On peut voir sur la figure 1.2 que $f^k(x) \geq f(x), \forall x \in C$ et $f^k(x^k) = f(x^k)$. DCA construit d'abord un suprémum $f^k(x)$ de la fonction $f(x)$ au point x^k, et ensuite minimise cette fonction $f^k(x)$ sur C afin d'obtenir le point x^{k+1}, et ainsi de suite pour obtenir le point x^{k+2}... On peut également retrouver une explication explicite

de beaucoup de propriétés importantes de DCA sur la figure (e.g. décroissance, bornitude, convergence, etc.).

Par exemple, théoriquement, DCA construit une suite $\{x^k\}$ telle que la suite $\{f(x^k)\}$ est décroissante. Ceci peut être facilement vérifié sur cette figure car x^{k+1} est un minimum de f^k (donc $f^k(x^k) \geq f^k(x^{k+1})$) et $f(x^{k+1}) \leq f^k(x^{k+1})$, ainsi $f^k(x^k) = f(x^k)$. Finalement, on a $f(x^k) \geq f^k(x^{k+1}) \geq f(x^{k+1})$. Ceci montre que la suite $\{f(x^k)\}$ est décroissante.

Pour la bornitude et la convergence de DCA, sachant que $f : \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$ est bornée inférieurement sur $\mathbb{R}^n$, la suite $\{f(x^k)\}$ est aussi bornée inférieurement. On sait qu'une suite $\{f(x^k)\}$ décroissante et bornée inférieurement est convergente.

Lorsque DCA converge vers un point x^*, ce point doit être un point KKT généralisé. Cela est aussi aisé à comprendre, à l'aide de la figure. Si on lance DCA à partir d'un point KKT généralisé de f (e.g. x^* sur la figure), alors DCA s'arrête immédiatement à ce point car il est aussi un minimum de f^* (la fonction majorante convexe définie au point x^* par $f^*(x) = g(x) - [h(x^*) + \langle x - x^*, y^* \rangle]$, $y^* \in \partial h(x^*)$).

Grâce à la figure, on peut constater que DCA a la faculté de sauter certains voisinages de minima locaux. Par exemple, DCA saute un minimum local entre x^k et x^{k+1} et parvient à un voisinage de la solution globale x^*. Ce phénomène est très intéréssant et important pour l'optimisation globale. Bien que l'on ne puisse pas toujours garantir que ce phénomène se produit, on peut comprendre que la performance de DCA est susceptible de dépendre de la decomposition DC et de la position du point initial. Par conséquent, pour appliquer DCA, il faut toujours penser aux deux questions suivantes :

1. Le choix du point initial x^0 ?
2. Le choix des composantes DC g et h ?

Actuellement, sur le plan théorique, il n'y a pas d'approche déterministe pour répondre à ces deux questions. Elles restent encore ouvertes. En pratique, on choisit des décompositions DC adaptées à la structure spécifique du problème DC donné. On préfère souvent une décomposition de la structure plus simple et moins coûteuse pour la construction, telle que les suites $\{x^k\}$ et $\{y^k\}$ soient facilement calculables. On cherche habituellement à ce que :

- La fonction h permette de calculer facilement ∂h.
- Le programme convexe $min\{g(x) - \langle x, y^k \rangle : x \in X\}$ soit facile à résoudre, cela est notamment important pour les problèmes de grande dimensions.
- Le point initial x^0 soit aussi proche que possible d'une solution optimale globale.

Il est très important de noter la richesse de la programmation DC et DCA, en particulier sur le plan algorithmique : Il a été démontré (dans [29]) qu'avec des décompositions DC convenables, on peut retrouver la plupart des algorithmes standards

en programmation convexe et non convexe. Dans la section suivante, un exemple très simple montre que l'algorithme de gradient projeté est un cas particulier de DCA dont on aura besoin dans le chapitre 6.

1.6.4 DCA et l'algorithme de Gradient Projeté

Considérons le problème d'optimisation suivant :

$$(P) \qquad \alpha = \inf\{f(x) : x \in C\}$$

où $f : \mathbb{R}^n \to \mathbb{R}$ est une fonction de classe $C^2(\mathbb{R}^n)$, et C un convexe fermé de $\mathbb{R}^n$. Soit une fonction quadratique convexe $g_\lambda(x) = \frac{\lambda}{2}\|x\|^2$ avec $\lambda > 0$. On peut choisir une décomposition DC $f(x) = g_\lambda(x) - [g_\lambda(x) - f(x)]$, où $g_\lambda(x) - f(x)$ et $g_\lambda(x)$ sont deux fonctions convexes sur C. Il est facile de vérifier qu'un tel nombre λ existe si et seulement si $\rho(\nabla^2 f(x)) < +\infty, \forall x \in C$, où $\rho(\nabla^2 f(x))$ est le rayon spectral de la matrice hessienne $\nabla^2 f(x)$. On peut donc choisir

$$\lambda = \min\{\lambda : \lambda > 0, \lambda \geq \rho(\nabla^2 f(x)), x \in C\}.$$

Dans ce cas, on dit que g_λ est plus convexe que f sur C.

Par conséquent, le problème (P) est équivalent au problème DC

$$(\bar{P}) \qquad \alpha = \inf\{f(x) = (g + \chi_C)(x) - h(x) : x \in \mathbb{R}^n\}$$

où $g(x) = \frac{\lambda}{2}\|x\|^2$; χ_C est la fonction indicatrice de C ; $h(x) = g(x) - f(x)$. $(\bar{P})$ est un programme DC car $(g + \chi_C)$ et h sont deux fonctions convexes sur $\mathbb{R}^n$. Sachant que g et f sont deux fonctions de classe C^2 alors

$$\partial h(x) = \nabla h(x) = \nabla g(x) - \nabla f(x) = \lambda x - \nabla f(x).$$

DCA pour résoudre ce problème particulier $(\bar{P})$, peut être décrit par le schéma de point fixe suivant :

$$x^{k+1} \in \arg\min\{(g + \chi_C)(x) - \langle x, \nabla h(x^k)\rangle : x \in \mathbb{R}^n\}.$$

Ceci équivaut à

$$x^{k+1} = P_C(x^k - \frac{\nabla f(x^k)}{\lambda}),$$

où P_C désigne l'opérateur de projection orthogonale sur C. On reconnaît l'algorithme de gradient projeté avec le pas $\frac{1}{\lambda}$.

1.6.5 Techniques de pénalité exacte

Les techniques de pénalité exacte sont souvent très utiles pour transformer un problème DC sous contraintes anti-convexes en un problème DC sous contraintes convexes. Ce dernier problème peut être résolu par DCA. Pour plus de détails sur les techniques de pénalité exacte pour la programmation DC, on pourra se référer à [26, **?**].

Considérons un problème d'optimisation

$$(P) \qquad \alpha = \inf\{f(x) : x \in C, p(x) \leq 0\}$$

où C est un polyèdre convexe non vide et borné dans $\mathbb{R}^n$, f, p sont deux fonctions concaves finies sur C, où p est une fonction à valeurs positives ou nulles sur C. Le problème (P) est un programme non convexe. La non convexité du problème est due au fait que la fonction objectif f est concave et que la contrainte $p(x) \leq 0$ est non convexe. Une contrainte $p(x) \leq 0$ avec p concave est dite anti-convexe. L'objectif du problème (P) est de minimiser une fonction concave f sous contraintes polyédrales (convexes) et une contrainte anti-convexe. Supposons que le problème (P), dont l'ensemble de solutions est noté par $\mathcal{P}$, est réalisable. Étant donné un paramètre $t > 0$, on peut définir le problème de pénalité par :

$$(P_t) \qquad \alpha(t) = \inf\{f(x) + tp(x) : x \in C\}.$$

L'ensemble des solutions du problème (P_t) est noté par $\mathcal{P}_t$. On obtient le théorème suivant :

Théorème 1.45 (Théorème de pénalité exacte) *Soit C un polyèdre convexe non vide et borné ; f et p sont deux fonctions concaves finies sur C, où p est une fonction à valeurs positives ou nulles sur C. Alors il existe un nombre fini $t_0 \geq 0$ tel que pour tout $t \geq t_0$, les deux problèmes (P) et (P_t) sont équivalents au sens que $\mathcal{P}_t = \mathcal{P}$ et $\alpha(t) = \alpha$. Le paramètre t_0 peut être déterminé comme suit :*

- *Si l'ensemble des sommets $V(C)$ de C est contenu dans l'ensemble $\{x \in C : p(x) \leq 0\}$, alors $t_0 = 0$ et $\alpha(0) = \alpha$.*
- *Si $\alpha(0) < \alpha$ alors $t_0 = \max\{\frac{\alpha - f(x)}{p(x)} : x \in V(C), p(x) > 0\}$*

On a aussi les propriétés suivantes :

- *$\alpha(t) = \alpha$ si et seulement si $t \geq t_0$*
- *$\mathcal{P}_t \cap \{x \in C : p(x) \leq 0\} \neq \emptyset \Longleftrightarrow \mathcal{P}_t \subset \mathcal{P} \Longleftrightarrow t \geq t_0$*
- *$\mathcal{P}_t = \mathcal{P}$ si $t > t_0$*

Ce théorème est souvent très utile en programmation non convexe, notamment en programmation DC, parce que beaucoup de problèmes d'optimisation sous contraintes non convexes peuvent être transformés en problème (P) à l'aide de ce théorème.

Remarquons que l'estimation du paramètre t n'est pas toujours évidente. En pratique, on donne souvent une valeur assez grande a priori pour simplifier l'estimation de t. Pour s'assurer que la valeur de t est assez grande, on propose le théorème suivant :

Théorème 1.46 *Soient C un polyèdre convexe non vide et borné, f et p deux fonctions concaves finies sur C, où p est une fonction à valeurs positives ou nulles sur C. Alors il existe un nombre fini $t^* > 0$ tel que pour tout $t \geq t^*$, les deux suites générées par DCA $\{f(x^k + tp(x^k)\}$ et $\{p(x^k)\}$ sont décroissantes.*

Ce théorème montre qu'il existe une valeur de t assez grande telle que la suite $\{p(x^k)\}$ générée par DCA soit décroissante. Autrement dit, si l'on trouve que la suite $\{p(x^k)\}$ n'est pas décroissante, alors t n'est pas assez grand. A l'aide de ce dernier théorème, on peut proposer une nouvelle approche modifiée de DCA pour résoudre le problème (P).

Algorithme DCA Modifié

Étape 0 : x^0 donné, t donné comme une grande valeur, $k = 0$

Étape 1 : On calcule $y^k \in \partial(-f - tp)(x^k)$.

Étape 2 : On détermine $x^{k+1} \in \arg\min\{-\langle x, y^k\rangle : x \in C\}$.

Étape 3 : Si $p(x^{k+1}) > p(x^k)$, alors on augmente la valeur de t. (e.g. : $t = t + 1000$)

Étape 4 : Si les conditions d'arrêt de DCA sont vérifiées, alors on termine DCA ; sinon $k = k + 1$ et on répète l'Étape 1.

L'idée de cette approche modifiée est d'augmenter la valeur de t à l'intération k si $p(x^{k+1}) > p(x^k)$.

Chapitre 2

Programmation semi-définie et Relaxation semi-définie

Sommaire

Dans cette partie, nous allons présenter quelques concepts fondamentaux et résultats essentiels en programmation semi-définie. Nous présentons des méthodes de reformulation semi-définie pour les programmes linéaires, les programmes quadratiques. Ensuite on présente les techniques de relaxation SDP, et leur application en programmation combinatoire. Les logiciels usuels pour résoudre un programme SDP sont aussi introduits dans ce chapitre. Pour plus de détails en programmation SDP et Relaxation SDP, on pourra se référer à [63, 64, 65, 66, 67].

2.1 Notations et définitions

Dans cette section, nous commençons par rappeler les notations et les définitions usuelles en programmation semi-définie. Notons $M_{m,n}$ l'espace vectoriel des matrices réelles $m \times n$, M_n l'espace des matrices carrées réelles $n \times n$ et S_n l'espace des matrices symétriques réelles $n \times n$.

Définition 2.1 *Une matrice $X \in S_n$ est dite semi-définie positive (resp. définie positive), notée $X \succeq O$ (resp. $X \succ O$), si et seulement si $u^T X u \geq 0, \forall u \in \mathbb{R}^n$ (resp. $u^T X u > 0, \forall u \in \mathbb{R}^n \setminus \{0\}$), où O désigne la matrice nulle $n \times n$.*

On note l'ensemble des matrices semi-définies positives (resp. définie positive) par S_n^+ (resp. S_n^{++}). L'ensemble des matrices semi-définies positives S_n^+ est un cône convexe. Nous avons la relation d'inclusion suivante :

$$S_n^{++} \subset S_n^+ \subset S_n \subset M_n.$$

Proposition 2.2 *$X \in S_n^+$ si et seulement si l'une des conditions suivantes est vérifiée :*

1. *les valeurs propres de X sont toutes positives ou nulles.*
2. *$u^T X u \geq 0, \forall u \in \mathbb{R}^n$.*
3. *Les déterminants de tous les mineurs symétriques de X sont positifs ou nuls.*

En effet, $M_{m,n}$ est identifié à l'espace Euclidien $\mathbb{R}^{m \times n}$ de dimension $m \times n$ et donc M_n est identifié à $\mathbb{R}^{n \times n}$ de dimension n^2. S_n est un sous-espace vectoriel de M_n de dimension $n(n+1)/2$. Il existe un opérateur linéaire qui permet de transformer une matrice $A \in M_{m,n}$ en un vecteur de $\mathbb{R}^{m \times n}$. Cet opérateur, noté `vect` est défini de la façon suivante :

$$\texttt{vect}(A) = \begin{bmatrix} A_{.,1} \\ A_{.,2} \\ \vdots \\ A_{.,n} \end{bmatrix}$$

où $A_{.,i}$ est la $i^{ème}$ colonne de la matrice A.

Une relation d'ordre partiel, l'ordre partiel Löwner, est défini sur S_n par :

$$A, B \in S_n, A \succeq B \ (A \succ B) \text{ si } A - B \in S_n^+ \text{ (resp. } S_n^{++}).$$

Définition 2.3 *Soient A, B deux matrices de $M_{m,n}$. Le produit scalaire habituel de A et B, noté $A \bullet B$, est défini comme la trace de la matrice $B^T A$. C'est à dire,*

$$A \bullet B = Tr(B^T A) = \sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij} B_{ij}.$$

En particulier, si $(A, B) \in S_n^2$, on a

$$A \bullet B = Tr(B^T A) = Tr(AB).$$

Avec ce produit scalaire, on peut réécrire toute forme quadratique $x^T A x + b^T x + c$ comme $A \bullet x x^T + b^T x + c$, où $x \in \mathbb{R}^n$.

Définition 2.4 *Soit l'application $F : \mathbb{R}^m \mapsto S_n$ définie par $F(x) = A_0 + \sum_{i=1}^m x_i A_i$, où $A_0, \ldots, A_m \in S_n$. Alors, une contrainte d'inégalité linéaire matricielle (**LMI**) est définie par*

$$\mathcal{S} = \{x \in \mathbb{R}^m : F(x) \succeq O\}.$$

$\mathcal{S}$ est un ensemble de vecteurs x tels que $F(x)$ est une matrice semi-définie positive. Si $F(x) \succeq O$ et $F(y) \succeq O$, alors, $\forall \lambda \in [0,1]$, on a

$$F(\lambda x + (1-\lambda)y) = \lambda F(x) + (1-\lambda)F(y) \succeq O.$$

Ceci démontre que $\mathcal{S}$ est un sous-ensemble convexe de $\mathbb{R}^m$.

Théorème 2.5 (Complément de Schur) *Soient $A \in S_p^{++}$, $B \in S_q$ et $C \in M_{p,q}$ alors :*

$$\begin{bmatrix} A & C \\ C^T & B \end{bmatrix} \succeq O \Longleftrightarrow B \succeq C^T A^{-1} C.$$

2.2 Programmation semi-définie et sa dualité

Un programme semi-défini (SDP) peut être considéré comme un programme linéaire sous une contrainte d'inégalité linéaire matricielle. Autrement dit, un programme linéaire avec un cône de matrices semi-définies positives.

Considérons le problème de minimisation d'une fonction linéaire d'une variable $x \in \mathbb{R}^m$ sous contrainte LMI :

$$\min\{c^T x : F(x) \succeq O\} \tag{2.1}$$

où $F(x) = A_0 + \sum_{i=1}^m x_i A_i$, $c \in \mathbb{R}^m$, $A_0, \ldots, A_m \in S_n$, et la matrice $\sum_{i=1}^m x_i A_i$ est appelée matrice de rigidité (stiffness matrix). Le problème dual associé au programme semi-défini (2.1) est

$$\max\{-A_0 \bullet Z : A_i \bullet Z = c_i, i = 1, \ldots, m; Z \succeq O\} \tag{2.2}$$

où $Z \in S_n$. Le problème dual (2.2) est un programme linéaire avec une variable matricielle semi-définie positive. Par analogie, supposons que le problème primal soit défini par :

$$\min\{A_0 \bullet X : A_i \bullet X = c_i, i = 1, \ldots, m; X \succeq O\}. \tag{2.3}$$

alors son problème dual peut s'écrire sous la forme

$$\max\{c^T y : \sum_{i=1}^m y_i A_i + M = A_0; M \succeq O\} \tag{2.4}$$

On peut aisément constater que le problème primal et son dual sont également des problèmes SDP. Il est facile de vérifier que le problème dual peut être reformulé sous la même forme que le problème primal. Un programme semi-défini (primal et dual) est un problème d'optimisation convexe. En effet, la fonction objectif du problème primal (resp. dual) est une fonction linéaire. Les contraintes consistent en des polyèdres convexes ainsi que des contraintes d'inégalités linéaires matricielles qui sont toutes convexes. Un SDP est donc un programme convexe. Les problèmes primal et dual

de SDP sont des problèmes de minimisation ou maximisation d'une fonction linéaire sous contraintes linéaires et sous contraintes matricielles semi-définies. La variable du problème peut être un vecteur ou une matrice.
Supposons que x est un point réalisable du problème primal (2.1) et que Z est un point réalisable du problème dual (2.2), nous avons

$$\eta = c^T x - (-A_0 \bullet Z) = \sum_{i=1}^{m} A_i \bullet Z x_i + A_0 \bullet Z = F(x) \bullet Z \geq 0,$$

du fait que $A, B \in S_n^+ \Longrightarrow A \bullet B \geq 0$. On a donc $-A_0 \bullet Z \leq c^T x$. En optimisation SDP, η est dit *saut de dualité* associé à x et Z. De façon analogue, on peut en déduire que le saut de dualité entre les solutions réalisables de problème primal (2.3) et de son dual (2.4) est

$$\eta_1 = A_0 \bullet X - c^T y = A_0 \bullet X - \sum_{i=1}^{m} A_i \bullet X y_i = (A_0 - \sum_{i=1}^{m} y_i A_i) \bullet Z = M \bullet X \geq 0.$$

On a alors le théorème de dualité faible :

Théorème 2.6 (Théorème de dualité faible) *Si on note p^* la valeur optimale du problème primal (2.1) ou (2.3) et d^* la valeur optimale du problème dual (2.2) ou (2.4), alors on a*

$$d^* \leq p^*.$$

Sous certaines hypothèses un peu plus fortes, nous avons un théorème de dualité forte pour les programmes SDP qui peut fournir des résultats aussi intéressants qu'en programmation linéaire.

Théorème 2.7 (Théorème de dualité forte) *Nous avons $d^* = p^*$ si l'une des conditions suivantes est vérifiée :*

- *Si le problème primal (2.1) est strictement réalisable (c.à.d., il existe un x tel que $F(x) \succ O$) alors son dual (2.2) a une solution optimale et $d^* = p^*$.*
- *Si le problème dual (2.2) est strictement réalisable (c.à.d., il existe un Z tel que $Z = Z^T \succ O, A_i \bullet Z = c_i, i = 1, \ldots, m$) alors son primal (2.1) a une solution optimale et $d^* = p^*$.*

Nous ne détaillerons pas la démonstration du théorème de dualité forte (voir les réferences [63, 64]).

Le théorème suivant décrit les conditions d'optimalité pour la programmation semi-définie :

Théorème 2.8 (Conditions d'Optimalité) *Supposons que les problèmes (2.1) et (2.2) sont strictement réalisables. Les points réalisables x et Z sont solutions optimales des problèmes primal (2.1) et dual (2.2) si et seulement si les conditions suivantes sont satisfaites :*

- $F(x) \succ O$.
- $Z = Z^T \succ O, A_i \bullet Z = c_i, i = 1, \ldots, m$.
- $F(x) \bullet Z = 0$.

2.3 Transformation d'un programme linéaire en un programme semi-défini

La programmation semi-définie peut être vue comme une généralisation de la programmation linéaire, puisque si toutes les matrices $A_i, i = 0, \ldots, m$ sont diagonales alors les problèmes primal et dual de SDP deviennent des programmes linéaires. Plus précisément, considérons un programme linéaire défini par :

$$\min\{c^T x : Ax = b, x \geq 0\} \tag{2.5}$$

où $A \in M_{m,n}$, $b \in \mathbb{R}^m$ et $x \in \mathbb{R}^n$. Pour transformer le programme linéaire (2.5) en un programme SDP, on peut définir

$$X = \mathbf{diag}(x) = \begin{bmatrix} x_1 & 0 & \cdots & 0 \\ 0 & x_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & x_n \end{bmatrix}$$

$$A_0 = \mathbf{diag}(c), \text{ et } A_i = \mathbf{diag}(A_{i,.}), i = 1, \ldots, m$$

où $A_{i,.}$ désigne la i^{me} ligne de la matrice A. Nous pouvons réécrire la fonction objectif linéaire

$$c^T x = A_0 \bullet X$$

et les contraintes linéaires

$$Ax = b \Longleftrightarrow A_i \bullet X = b_i, i = 1, \ldots, m.$$

Les contraintes $x \geq 0$ signifient que $X \succeq O$ et $X_{ij} = 0$ pour tous les élements de X hors diagonal.

Ainsi, le programme linéaire (2.5) de la variable x peut être réécrit sous la forme d'un programme SDP de la variable matricielle X comme suit :

$$\begin{array}{ll} min & A_0 \bullet X \\ s.c. & A_i \bullet X = b_i, i = 1, \ldots, m, \\ & X_{ij} = 0, i = 1, \ldots, n-1, j = i+1, \ldots, n, \\ & X \succeq O. \end{array} \tag{2.6}$$

En pratique, personne ne veut transformer un programme linéaire en un programme SDP pour la résolution numérique, mais du point de vue théorique, un programme linéaire peut être considéré comme un cas particulier de SDP.

2.4 Transformation d'un programme quadratique convexe en un programme semi-défini

Un problème de programmation quadratique convexe sous contraintes quadratiques convexes est défini par :

$$\begin{array}{ll} min & x^T A_0 x + b_0^T x + c_0 \\ s.c. & x^T A_i x + b_i^T x + c_i \leq 0, i = 1, \ldots, m, \end{array} \tag{2.7}$$

où toutes les matrices $A_i, i = 0, \ldots, m$ sont dans S_n^+, et la variable x est dans $\mathbb{R}^n$. On sait qu'un SDP est un programme linéaire avec des contraintes matricielles. Pour la programmation quadratique convexe, nous pouvons représenter les fonctions quadratiques convexes et les contraintes quadratiques convexes sous forme d'inégalités matricielles. Plus précisément, pour la fonction objectif, nous pouvons d'abord ajouter une variable auxiliaire réelle y pour obtenir une forme équivalente

$$\begin{array}{ll} min & y \\ s.c. & x^T A_0 x + b_0^T x + c_0 \leq y, \\ & x^T A_i x + b_i^T x + c_i \leq 0, i = 1, \ldots, m, \end{array} \tag{2.8}$$

où la fonction objectif devient linéaire et les contraintes restent quadratiques convexes. Ensuite, on peut transformer les contraintes quadratiques convexes sous forme matricielle semi-définie à l'aide du théorème de Schur. On souligne l'importance du théorème de Schur (le théorème 2.5) qui est souvent utile pour transformer une contrainte définie à l'aide de fonctions nonlinéaires en contrainte du type inégalité matricielle linéaire. Pour ce faire, nous rappelons la proposition suivante :

Proposition 2.1 *Toute matrice $A \in S_n^+$ admet une unique racine carrée $M \in S_n^+$, c.à.d., $A = M^2$.*

Supposons que $A_i = M_i^2, i = 0, \ldots, m$. En vertu du théorème de Schur, on a

$$\begin{bmatrix} I & M_i x \\ x^T M_i^T & -c_i - b_i^T x \end{bmatrix} \succeq O \Longleftrightarrow x^T A_i x + b_i^T x + c_i \leq 0$$

Cette dernière formule nous permet de transformer les contraintes quadratiques convexes en contraintes matricielles linéaires semi-définies. Le problème 2.9 peut se transformer :

$$\begin{array}{ll} min & y \\ s.c. & \begin{bmatrix} I & M_0 x \\ x^T M_0^T & -c_i - b_i^T x + y \end{bmatrix} \succeq O \\ & \begin{bmatrix} I & M_i x \\ x^T M_i^T & -c_i - b_i^T x \end{bmatrix} \succeq O, i = 1, \ldots, m. \end{array} \tag{2.9}$$

Remarque 6 Dans la formulation (2.9), il y a n+1 variables inconnues (x, y) et m+1 contraintes inégalité linéaires matricielles.

Considérons la proposition suivante :

Proposition 2.2 *Soient $x \in \mathbb{R}^n$ et $W \in S_n$, alors*

$$\begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O \text{ si et seulement si } W \succeq xx^T.$$

La démonstration est immédiate en utilisant le théorème de Schur. En vertu de cette proposition, nous pouvons simplement représenter le problème (2.9) sous la forme d'un problème SDP

$$\begin{array}{ll} min & y \\ s.c. & \begin{bmatrix} c_0 - y & \frac{1}{2}b_0^T \\ \frac{1}{2}b_0 & A_0 \end{bmatrix} \bullet \begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \leq 0 \\ & \begin{bmatrix} c_i & \frac{1}{2}b_i^T \\ \frac{1}{2}b_i & A_i \end{bmatrix} \bullet \begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \leq 0, i = 1, \ldots, m, \\ & \begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O. \end{array} \tag{2.10}$$

Dans la formulation (2.10), nous avons $\frac{(n+1)(n+2)}{2}$ variables et $m+1$ contraintes linéaires. En particulier, la plupart des contraintes du problème (2.10) sont linéaires, elles sont différentes de celles de la formulation (2.9) qui sont LMI. Du point de vue de calcul scientifique, cette dernière formulation peut être plus facile à résoudre.

Remarquons que si la forme quadratique est nonconvexe, c.à.d., la matrice A_i n'est pas semi-définie positive, alors A_i n'admet pas de racine carrée. La formulation (2.9) n'existe plus dans le cas nonconvexe. En revanche, la formulation (2.10) existe toujours, mais elle n'équivaut plus à (2.7) car elle devient un problème de borne inférieure. Cette formulation SDP s'appelle *rélaxation semi-définie*. Ce sujet sera discuté dans la Section 2.5.

Pour finir, on vérifie également que la programmation quadratique convexe sous contraintes quadratiques convexes est un cas particulier de la programmation semi-définie.

Le cône d'ordre deux et l'inégalité matricielle linéaire

Un cône d'ordre deux (*second-order cone*, SOC) est défini comme :

$$\|Ax + b\|_2 \leq c^T x + d$$

où $\|.\|_2$ est la norme euclidienne d'ordre deux, $A \in M_{m,n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$ et $d \in \mathbb{R}$ sont des paramètres donnés, $x \in \mathbb{R}^n$ est une variable.
Nous pouvons facilement vérifier les propriétés suivantes :

- Si $A = 0$ alors SOC devient une contrainte inégalité linéaire.
- Si $c = 0$ alors SOC est équivalent à une contrainte quadratique convexe.

Ce genre de contrainte peut être réécrit directement sous la forme de LMI

$$\|Ax+b\|_2 \leq c^T x + d \Longleftrightarrow \begin{bmatrix} (c^T x+d)I & Ax+b \\ (Ax+b)^T & c^T x+d \end{bmatrix} \succeq O$$

à l'aide du théorème de Schur.

2.5 Techniques de Relaxation Semi-définie

Nous avons montré qu'un programme convexe défini par une fonction objectif linéaire ou quadratique convexe sous contraintes linéaires, contraintes quadratiques convexes, ou contraintes de cône d'ordre deux peut être reformulé en un programme SDP équivalent. Cependant, trouver une reformulation SDP équivalente au programme non convexe est généralement impossible. La raison est naturellement logique, car un problème non convexe ne peut pas être transformé en un problème convexe sauf dans des cas très particuliers. Dans cette section, on se consacre aux techniques de représentation des contraintes quadratiques non convexes à l'aide de matrices semi-définies, ainsi qu'aux techniques de relaxation semi-définie.

2.5.1 Représentation Semi-définie

Pour représenter une contrainte quadratique non convexe $x^T Ax + b^T x + c \leq 0$ avec des matrices semi-définies, on introduit préalablement une nouvelle variable matricielle $W \succeq O$ telle que $W = xx^T$. On obtient alors le théorème suivant :

Théorème 2.9

$$W = xx^T \Longleftrightarrow \begin{cases} \begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O, \\ Tr(W) - \|x\|^2 \leq 0 \end{cases}$$

Preuve.

$$W = xx^T \Longleftrightarrow W - xx^T \succeq O \text{ et } W - xx^T \preceq O$$

Grâce au théorème de Schur, on a $W - xx^T \succeq O \Longleftrightarrow \begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O$. De plus, $W - xx^T \preceq O \Longrightarrow Tr(W - xx^T) = Tr(W) - Tr(xx^T) = Tr(W) - \|x\|^2 \leq 0$. Par conséquent, on a démontré que $W = xx^T \Longrightarrow \begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O$ et $Tr(W) - \|x\|^2 \leq 0$. Inversement, $\begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O$ et $Tr(W) - \|x\|^2 \leq 0 \Longleftrightarrow W - xx^T \succeq O$ et $Tr(W - xx^T) \leq 0$.

L'inégalité $W - xx^T \succeq O$ signifie que toutes les valeurs propres de la matrice $W - xx^T$ sont positives. De plus, $Tr(W - xx^T) \leq 0$ signifie que la somme de toutes les valeurs propres de $W - xx^T$ est négative ou nulle. Les deux formules sont vraies en même temps si et seulement si toutes les valeurs propres de $W - xx^T$ sont nulles. Cela preuve que $W = xx^T$. □

Le théorème 2.9 donne une représentation équivalente de la contrainte non convexe $W = xx^T$ dont $\begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O$ est un LMI (contrainte convexe) et $Tr(W) - \|x\|^2 \leq 0$ une contrainte anti-convexe. A l'aide du Théorème 2.9, la contrainte quadratique nonconvexe $x^T Ax + b^T x + c \leq 0$ peut être représentée via des contraintes linéaires, des contraintes matricielles linéaires, ainsi qu'une contrainte anti-convexe comme suit :

$$x^T Ax + b^T x + c \leq 0 \Longleftrightarrow \begin{cases} A \bullet W + b^T x + c \leq 0, \\ \begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O, \\ Tr(W) - \|x\|^2 \leq 0 \end{cases}$$

2.5.2 Relaxation de contrainte des variables binaires

Considérons une contrainte des variables binaires $x \in \{0,1\}^n$. Évidemment, les contraintes $x_i \in \{0,1\}, i = 1, \ldots, n$ sont équivalentes aux contraintes quadratiques d'égalité $x_i^2 = x_i, i = 1, \ldots, n$. Supposons que $W = xx^T$, alors

$$x_i^2 = x_i, i = 1, \ldots, n \Longleftrightarrow \begin{cases} diag(W) = x, \\ W = xx^T. \end{cases} \Longleftrightarrow \begin{cases} diag(W) = x, & (1) \\ \begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O, & (2) \\ Tr(W) - \|x\|^2 \leq 0. & (3) \end{cases}$$

La plupart des travaux sur la relaxation semi-définie vont éliminer directement la contrainte anti-convexe (3) afin d'obtenir un ensemble relaxé convexe $\{(1), (2)\}$. En effet, pour les variables binaires, ce genre de relaxation équivaut à la relaxation linéaire standard $x \in [0,1]^n$ car on peut facilement vérifier que

$$\forall (x, W) \in \{(1), (2)\} \Longrightarrow (x, W) \in \{(x, W) : \begin{bmatrix} 1 & x_i \\ x_i & W_{ii} \end{bmatrix} \succeq O, W_{ii} = x_i, i = 1, \ldots, n\}$$

$$\Longrightarrow W_{ii} - x_i^2 \geq 0, W_{ii} = x_i, i = 1, \ldots, n \Longrightarrow x_i - x_i^2 \geq 0, i = 1, \ldots, n$$

C'est à dire $x \in [0,1]^n$. Inversement, quel que soit $x \in [0,1]^n$, il existe une matrice $W := diag(x)$ telle que $(x, W) \in \{(1), (2)\}$. Autrement dit, la projection de l'ensemble convexe $\{(1), (2)\}$ dans l'espace de x est exactement le rectangle $[0,1]^n$.

Considérons l'enveloppe convexe $co_{[0,1]^n}(Tr(W) - \|x\|^2)$ défini sur le rectangle $x \in [0,1]^n$. On a $co_{[0,1]^n}(Tr(W) - \|x\|^2) = Tr(W) - \sum_{i=1}^n (x_i)$ et donc

$$Tr(W) - \sum_{i=1}^{n} (x_i) = co_{[0,1]^n}(Tr(W) - \|x\|^2) \leq Tr(W) - \|x\|^2 \leq 0, \forall x \in [0,1]^n. \qquad (4)$$

L'ensemble $\{(1), (2), (4)\}$ est une relaxation convexe des contraintes $\{(1), (2), (3)\}$. Par contre, la contrainte (4) sera éliminée quand la contrainte (1) est présente. Donc $\{(1), (2), (4)\}$ équivaut à $\{(1), (2)\}$.

On montre que pour un problème d'optimisation linéaire sans contrainte avec des variables binaires, la relaxation SDP est équivalente à la relaxation linéaire standard. C.à.d, la relaxation SDP n'est pas pertinente pour ce genre de problème sans contrainte. Par contre, l'utilisation de la relaxation SDP est intéressante avec la présentation de contraintes linéaires ou/et de contraintes nonlinéaires. Considérons le problème d'optimisation suivant :

$$\begin{array}{ll} min & c^T x \\ s.c. & a^T x \leq b \\ & x \in \{0,1\}^n. \end{array} \tag{2.11}$$

A l'aide du théorème 2.5.1, on trouve le problème équivalent

$$\begin{array}{ll} min & c^T x \\ s.c. & a^T x \leq b \\ & diag(W) = x \\ & \begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O \\ & Tr(W) - \|x\|^2 \leq 0. \end{array} \tag{2.12}$$

Remplaçant x par $diag(W)$, on obtient alors

$$\begin{array}{ll} min & c^T(diag(W)) \\ s.c. & a^T(diag(W)) \leq b \\ & \begin{bmatrix} 1 & diag(W)^T \\ diag(W) & W \end{bmatrix} \succeq O \\ & Tr(W) - \|diag(W)\|^2 \leq 0. \end{array}$$

Cette dernière formulation peut être écrite avec les notations de SDP sous la forme :

$$\begin{array}{ll} min & diag(c) \bullet W \\ s.c. & diag(a) \bullet W \leq b \\ & \begin{bmatrix} 1 & diag(W)^T \\ diag(W) & W \end{bmatrix} \succeq O \\ & Tr(W) - \|diag(W)\|^2 \leq 0. \end{array} \tag{2.13}$$

Le problème (2.13) est équivalent au problème (2.11) dans le sens où si (x, W) est une solution optimale du problème (2.13) alors x est une solution optimale du problème (2.11). Évidemment, le problème (2.13) est non convexe car la contrainte $Tr(W) - \|diag(W)\|^2 \leq 0$ est anti-convexe. Si on élimine cette contrainte, on obtient le problème de relaxation SDP

$$\begin{array}{ll} min & diag(c) \bullet W \\ s.c. & diag(a) \bullet W \leq b \\ & \begin{bmatrix} 1 & diag(W)^T \\ diag(W) & W \end{bmatrix} \succeq O. \end{array} \tag{2.14}$$

Le problème de relaxation SDP (2.14) est un programme SDP qui n'est pas équivalent au problème (2.11) mais sa valeur optimale est une borne inférieure de la valeur optimale du problème (2.11). Sans perte de généralité, on peut supposer que $0 \leq a_i, i = 1, \ldots, n$. En effet, si $a_i < 0$, on remplace la variable x_i par $1 - x_i$.
Plus concrètement, $\forall x \in \{0,1\}^n$, on peut créer une variable y telle que $y_i = x_i$ si $a_i \geq 0$; $y_i = 1 - x_i$ si $a_i < 0$. On a donc $\forall x \in \{0,1\}^n \Rightarrow y \in \{0,1\}^n$ (ainsi $\forall x \in [0,1]^n \Rightarrow y \in [0,1]^n$). Dans ce cas, on obtient

$$\begin{array}{ll} \sum_{i=1}^n a_i x_i & = \sum_{a_i \geq 0} |a_i| x_i + \sum_{a_i < 0} |a_i|(1 - x_i) - \sum_{a_i < 0} |a_i| \\ & = \sum_{i=1}^n |a_i| y_i - \sum_{a_i < 0} |a_i|. \end{array}$$

Par cette raison, la contrainte linéaire $a^T x \leq b, x \in \{0,1\}^n$ équivaut à $|a|^T y \leq b + \sum_{a_i<0} |a_i|, y \in \{0,1\}^n$ (également équivalent si $x \in [0,1]^n$). On peut donc toujours supposer que $a \geq 0$, et on obtient

$$|a^T x| = a^T x \leq b, x \in [0,1]^n.$$

On peut ensuite remplacer la contrainte $a^T x \leq b$ par la contrainte $|a^T x| \leq b$, et déduire plusieurs problèmes de relaxation SDP à l'aide de la variable matricielle. Par exemple :
Supposons que $x \in \{0,1\}^n$, $W = xx^T, diag(W) = x$, et $a \geq 0$ alors

1. $$|a^T x| \leq b \Longrightarrow a^T x x^T a \leq b^2 \Longleftrightarrow (aa^T) \bullet W \leq b^2.$$

2. $$|a^T x| \leq b \Longrightarrow 0 \leq |a^T x|(b - |a^T x|) \Longrightarrow 0 \leq a^T x(b - a^T x) = (0, a^T) \begin{bmatrix} 1 & x^T \\ x & xx^T \end{bmatrix} \begin{pmatrix} b \\ -a \end{pmatrix}$$

$$\Longleftrightarrow \begin{bmatrix} \begin{pmatrix} b \\ -a \end{pmatrix} & \begin{pmatrix} 0 \\ a \end{pmatrix}^T \end{bmatrix} \bullet \begin{bmatrix} 1 & diag(W)^T \\ diag(W) & W \end{bmatrix} \geq 0.$$

3. $|a^T x| = a^T x \leq b, x_i \geq 0, 1 - x_i \geq 0, i = 1, \ldots, n \Longrightarrow x_i a^T x \leq b x_i, (1 - x_i) a^T x \leq b(1 - x_i), i = 1, \ldots, n \Longrightarrow \sum_{j=1}^n a_i W_{i,j} \leq b W_{i,j}, i = 1, \ldots, n.$

Les trois formulations viennent d'une idée très simple selon laquelle la multiplication de deux inégalités linéaires non négatives valides conduit à une inégalité quadratique valide (Helmberge [68], Bauvin et Geomans [71], Balas [69], Sherali et Adams [87], Lovasz et Shrijver [72] etc.). C'est une technique de représentation carrée (square representation) pour une contrainte d'inégalité linéaire. On en déduit trois formulations de problème de relaxation SDP :
Représentation carrée (Helmberg) :

$$\begin{array}{ll} min & diag(c) \bullet W \\ s.c. & (aa^T) \bullet W \leq b^2 \\ & \begin{bmatrix} 1 & diag(W)^T \\ diag(W) & W \end{bmatrix} \succeq O \end{array} \tag{2.15}$$

Représentation carrée étendue (Bauvin et Geomans) :

$$\begin{array}{ll} min & diag(c) \bullet W \\ s.c. & \begin{bmatrix} \begin{pmatrix} b \\ -a \end{pmatrix} & \begin{pmatrix} 0 \\ a \end{pmatrix}^T \end{bmatrix} \bullet \begin{bmatrix} 1 & diag(W)^T \\ diag(W) & W \end{bmatrix} \geq 0 \\ & \begin{bmatrix} 1 & diag(W)^T \\ diag(W) & W \end{bmatrix} \succeq O \end{array} \tag{2.16}$$

Représentation carrée 2 :

$$\begin{array}{ll} min & diag(c) \bullet W \\ s.c. & \sum_{j=1}^{n} a_i W_{i,j} \leq b W_{i,i}, i = 1, \ldots, n \\ & \begin{bmatrix} 1 & diag(W)^T \\ diag(W) & W \end{bmatrix} \succeq O \end{array} \tag{2.17}$$

Remarque 7 Il est facile de vérifier que si l'on ajoute la contrainte anti-convexe $Tr(W) - \|diag(W)\|^2 \leq 0$ *aux trois relaxations, alors on obtient un problème équivalent à* (2.11).

En comparaison avec la première relaxation (2.14), appelée répresentation diagonale (diagonal representation), on a le lemme suivant :

Lemme 2.10 (Helmberge et al. [68]) *Soit l'ensemble réalisable du problème (2.13) (resp. (2.14), (2.15), (2.16) et (2.17)) noté* $\mathfrak{X}_0$ *(resp.* $\mathfrak{X}_1$*,* $\mathfrak{X}_2$*,* $\mathfrak{X}_3$ *et* $\mathfrak{X}_4$*), alors*

$$\mathfrak{X}_1 \supseteq \mathfrak{X}_2 \supseteq \mathfrak{X}_3 \supseteq \mathfrak{X}_4 \supseteq \mathfrak{X}_0.$$

Helmberge [65] a aussi trouvé un exemple spécifique pour montrer que les écarts entre les relaxations SDP de (2.14) à (2.17) peuvent être très grands.

En pratique, on préfère choisir une relaxation SDP facile à construire, dont l'ensemble réalisable est le plus petit possible, de manière à résoudre efficacement le problème de relaxation. Dans le contexte de l'optimisation globale, la relaxation SDP est souvent utilisée pour établir un problème de la borne inférieure pour un problème de minimisation. Beaucoup de résultats dans l'application de la relaxation SDP en optimisation combinatoire montrent qu'elle fournit souvent une borne inférieure de bonne qualité qui est beaucoup mieux que la relaxation linéaire.

2.6 Logiciels pour résoudre le programme semi-défini

Dans cette section, on présente les outils ("solveurs") usuels et connus utilisés dans nos travaux pour la résolution numérique du programme SDP, on présente dans ce chapitre que l'on utilise au cours de nos travaux de recherche. Ces logiciels sont souvent gratuits pour les recherches académiques et téléchargeable sur le site de

l'auteur.

SDPA :
SDPA [83] est une implémentation de la méthode primal-dual du point intérieur (PD-IPM) développée en langage C/C++ et MATLAB par le groupe de Kojima. Ce logiciel consiste en diverses versions selon les besoins d'utilisation. SDPA-GMP (SDPA with arbitrary precision arithmetic) ; SDPA-QD (SDPA with pseudo quad-double precision arithmetic) ; SDPA-DD (SDPA with pseudo double-double precision arithmetic) ; SDPA-M (SDPA with MATLAB interface) ; SDPARA (SDPA parallel version) ; SDPA-C (SDPA with the positive definite matrix Completion) ; SDPARA-C (parallel version of the SDPA-C). Pour plus de détails, vous pouvez consulter le site http ://sdpa.indsys.chuo-u.ac.jp/sdpa/. Il existe aussi une version de SDPA en ligne (online solver) sur le site http ://laqua.indsys.chuo-u.ac.jp/portal/. Il est possible d'uploader le fichier associe au problème SDP (au format SDPA) sur ce site et d'obtenir le résultat numérique immédiatement.

CSDP :
CSDP est une implémentation de PD-IPM développée en langage C par Borchers [73]. Il existe également une version de MATLAB. Ce logiciel est compatible avec Linux, Windows et Mac OS. Pour plus de détails, vous pouvez se référer au site https ://projects.coin-or.org/Csdp/.

DSDP :
DSDP est une implémentation de l'algorithme dual de point intérieur développée en langage C par Steven J. Benson, Ye Yinyu et al. [74]. Il est plutôt conçu pour les problèmes SDP formulés avec des matrices très creuses qui ont des structures particulières comme pour l'application en optimisation combinatoire. Une interface MATLAB, une bibliothèque appelable et sa version parallèle sont disponibles. Pour plus de détails, voir le site http ://www.mcs.anl.gov/hs/software/DSDP/.

SBMethod :
C'est une implémentation de la méthode spectrale bundle développée en langage C++ par Helmberg et al. Ce logiciel a pour but de résoudre le problème de minimisation de la valeur propre maximale d'une fonction affine matricielle. Il est possible de résoudre un problème de grande dimension. Pour plus de détails, vous pouvez vous référer au site http ://www-user.tu-chemnitz.de/ helmberg/SBmethod/.

SDPLR :
SDPLR réformule le problème SDP en un programme non linéaire et ensuite résolu via une méthode du lagrangien augmenté. Elle est recommandée pour résoudre les problèmes de grande dimension qui ne nécessitent pas une grande précision de solution. Pour plus de détails, vous pouvez vous référer au site http ://dollar.biz.uiowa.edu/ sburer/www/doku.php ?id=software#sdplr.

SDPT3 :
SDPT3 est une implémentation de PD-IPM en MATLAB développée par Toh et Todd. Il peut aussi résoudre les programmes coniques d'ordre deux et les programmes linéaires. Consultez le site http ://www.math.nus.edu.sg/ mattohkc/sdpt3.html pour savoir au plus.

SeDuMi :
SeDuMi est aussi une application de la DP-IPM en MATLAB développée par Sturm. Il est possible de résoudre des problèmes d'optimisation de variables réelles et de variables complexes de grande dimension. Les programmes coniques du second ordre et les programmes linéaires sont aussi compatibles. La caractéristique remarquable de ce logiciel est sa stabilité et son efficacité. Pour plus de détails, consultez le site http ://sedumi.ie.lehigh.edu/.

PENNON :
PENNON est un logiciel permettant de résoudre les programmes convexes et les programmes non convexes, y compris les problèmes de SDP nonlinéaires et les problèmes de BMI. Il s'agit d'une implémentation de la méthode du lagrangien augmenté développée par Stingl et Kocvara. C'est un logiciel commercial qui a été intégré dans TOMLAB. Pour plus de détails, vous pouvez vous référer au site http ://www2.am.uni-erlangen.de/ kocvara/pennon/.

YALMIP :
C'est un solveur sur MATLAB développé par Löfberg [102]. Ce logiciel fournit une interface pour faciliter l'utilisation des solveurs externes pour résoudre des problèmes d'optimisation (tels que le programme linéaire, quadratique, cônique du second ordre, semi-défini, conique mixte en variables entières, géométrique, polynomial, multiparamétrique ...). Ce logiciel est compatible avec tous les logiciels mentionnés ci-dessus, et aussi avec beaucoup d'autres logiciels gratuits ou commerciaux, comme GLPK, CLP, MOSEK, CPLEX, Xpress, GAMS etc. Pour plus de détails, vous pouvez consulter le site http ://control.ee.ethz.ch/ joloef/wiki/pmwiki.php.

Chapitre 3

Techniques de Séparation et Evaluation

Sommaire

Un algorithme de Séparation et Evaluation Progressive (SEP) ou simplement Séparation et Evaluation (SE) en français, Branch-and-Bound (B&B) en anglais, est une technique d'optimisation globale générale pour la résolution des problèmes d'optimisation non convexes, et plus particulièrement des problèmes d'optimisation combinatoire.

Considérons un problème d'optimisation

$$(P) \qquad \min\{f(x) : x \in S\},$$

où $S \subset \mathbb{R}^n$ est l'ensemble des solutions réalisables qui est compact non vide (pas nécessairement convexe). Supposons que la fonction f est finie sur S. Il est bien connu que le problème (P) admet une solution optimale globale $x^* \in S$ telle que

$$f(x^*) \leq f(x), \forall x \in S.$$

Le but de la méthode SE est de trouver cette solution x^*. L'idée de base de la méthode SE consiste en deux phases : "Séparation" et "Evaluation progressive".

1. La phase de séparation consiste à diviser successivement l'ensemble S en sous-ensembles qui forment une partition de l'ensemble S. On a donc le problème d'optimisation associé à chaque sous-ensemble. Par conséquent, si on résout tous les sous-problèmes et que l'on prend la meilleure solution trouvée, alors on est assuré d'avoir résolu le problème initial.
2. La phase d'évaluation progressive a pour but de déterminer l'optimum de ces sous-problèmes. Pour un sous-problème donné, son optimum peut être déterminé lorsque ce problème est facile à résoudre. Par exemple, lorsque le sous-ensemble réduit au singleton, le sous-problème est effectivement simple car l'optimum est l'élément unique de cet ensemble. Néanmoins, en pratique, on rencontre souvent un sous-problème qui est non convexe et difficile à résoudre. La méthode la plus générale consiste à déterminer une borne inférieure pour le coût des solutions contenues dans ce sous-ensemble (s'il s'agit d'un problème de minimisation). Si on arrive à trouver une borne inférieure qui est supérieure au coût de la meilleure solution réalisable (la borne supérieure) trouvée jusqu'à présent, on a alors l'assurance que le sous-ensemble ne contient pas l'optimum. On peut ensuite éliminer ce sous-ensemble et sélectionner un autre ensemble dans lequel la phase de séparation se poursuit. Les techniques les plus classiques pour le calcul de bornes sont basées sur l'idée de relaxation : relaxation continue, relaxation lagrangienne, relaxation SDP, etc.

Définition 3.1 *Soient M un compact dans $\mathbb{R}^n$ et I un ensemble fini d'indices. Un ensemble $M_i : i \in I$ de sous-ensembles compacts est dit une partition de M si*

$$M = \bigcup_{i \in I} M_i, M_i \cap M_j = \partial M_i \cap \partial M_j, \forall i, j \in I : i \neq j$$

où ∂M_i désigne la frontière relative à M de M_i.

3.1 Méthode de Séparation et Evaluation Progressive

Le schéma général de SE pour le problème (P) se résume de la manière suivante :

3.1.1 Prototype de SE [75]

- Initialisation
  1. Choisir un compact $S \subset M_0$, un ensemble fini d'indices I_0, et une partition $\mathcal{M}_0 = \{M_{0,i} : i \in I_0\}$ de M_0 satisfaisant $M_{0,i} \cap S \neq \emptyset, i \in I_0$.
  2. Pour chaque $i \in I_0$, déterminer

$$S_{0,i} \subset M_{0,i} \cap S, S_{0,i} \neq \emptyset$$

  et la borne supérieure

$$\alpha_{0,i} = \alpha(M_{0,i}) = \min f(S_{0,i}), x^{0,i} \in \arg\min f(S_{0,i}).$$

3. Pour chaque $i \in I_0$, déterminer la borne inférieure

$$\beta_{0,i} = \beta(M_{0,i}) \leq \min f(S \cap M_{0,i}).$$

4. Calculer

$$\alpha_0 = \min_{i \in I_0} \alpha_{0,i}$$

$$x^0 \in \arg\min f(x^{0,i}), i \in I_0$$

$$\beta_0 = \min_{i \in I_0} \beta_{0,i}.$$

- Itération $k = 0, 1, \ldots$

Étape1 : Supprimer tout $M_{k,i} \in \mathcal{M}_k$ vérifiant

$$\beta_{k,i} \geq \alpha_k$$

ou pour lequel on sait que $\min f(S)$ ne peut pas avoir lieu dans $M_{k,i}$. Soit L_k la liste des éléments restants $M_{k,i} \in \mathcal{M}_k$.
Si $L_k = \emptyset$ **alors s'arrêter** : x^k est une solution.

Étape2 : Sélectionner une partition $M_{k,i} \in L_k$ et choisir un ensemble fini d'indices J_{k+1} puis construire une partition pour $M_{k,i}$

$$\mathcal{M}_{k,i} = M_{k+1,i} : i \in J_{k+1}$$

de $M_{k+1,i}$ telle que $M_{k+1,i} \cap S \neq \emptyset$.

Étape3 : Pour chaque $i \in J_{k+1}$, déterminer les bornes supérieures sur $M_{k+1,i}$,

$$S_{k+1,i} \subset M_{k+1,i} \cap S, S_{k+1,i} \neq \emptyset$$

et la borne supérieure

$$\alpha_{k+1,i} = \alpha(M_{k+1,i}) = \min f(S_{k+1,i}), x^{k+1,i} \in \arg\min f(S_{k+1,i}).$$

Étape4 : Pour chaque $i \in J_{k+1}$, déterminer la borne inférieure sur $M_{k+1,i}$.,

$$\beta_{k+1,i} = \beta(M_{k+1,i}) \leq \min f(S \cap M_{k+1,i}).$$

Étape5 : Poser

$$\mathcal{M}_{k+1} = (L_k \cap M_{k,i}) \cup \mathcal{M}_{k,i}$$

Soit I_{k+1} l'ensemble d'indices tels que

$$M_{k+1} = \{M_{k+1,i} : i \in I_{k+1}\}$$

est la partition actuelle.

Étape6 : Calculer

$$\alpha_{k+1} = \min_{i \in I_{k+1}} \alpha_{k+1,i}$$

$$x^{k+1} \in \arg\min f(x^{k+1,i}), i \in I_{k+1}$$

$$\beta_{k+1} = \min_{i \in I_{k+1}} \beta_{k+1,i}.$$

$k = k + 1$ et répéter l'Itération.

Remarque 8 1. *Il faut déterminer $S_{k,i}$, $x^{k,i}$ et $\beta_{k,i}$ de telle façon que les bornes associées soient autant serrées que possible, avec un effort de calcul raisonnable.*

2. *$\alpha_{k,i}$ et $\beta_{k,i}$ sont des bornes supérieures et des bornes inférieures pour* $\min f(S \cap M_{k,i})$ *associées à chaque ensemble $M_{k,i}$, et α_k et β_k sont des bornes supérieures et des bornes inférieures pour* $\min f(S)$*, décroissantes et croissantes respectivement.*

3. *$\beta_{k,i} \geq \alpha_k$ indique que la solution actuelle $x^{k,i}$ ne peut pas s'améliorer dans $M_{k,i}$ donc cet ensemble peut être éliminé.*

3.1.2 Conditions de convergence

La méthode SE converge dans le sens où chaque point d'accumulation de $\{x^k\}$ est une solution de (P). Évidemment, par la construction de SE, on a

$$x^k \in S, k = 0, 1, \ldots$$

$$\alpha_k \geq \alpha_{k+1} \geq \min f(S) \geq \beta_{k+1} \geq \beta_k$$

$$f(x^k) \geq f(x^{k+1}), k = 0, 1, \ldots$$

Définition 3.2 *Une estimation de borne est dite cohérente (en anglais, consistent) si pour une suite décroissante M_{k_q,i_q} générée par la procédure de séparation telle que $M_{k_{q+1},i_{q+1}} \subset M_{k_q,i_q}$, on a*

$$\lim_{q \to \infty} (\alpha_{k_q,i_q} - \beta_{k_q,i_q}) = 0$$

Puisque $\beta_{k_q,i_q} \leq \alpha_{k_q} \leq \alpha_{k_q,i_q}$, la condition de la convergence peut s'écrire

$$\lim_{q \to \infty} (\alpha_{k_q} - \beta_{k_q,i_q}) = 0$$

Par la monotonie et la bornitude des suites $\{\alpha_k\}$ et $\{\beta_k\}$ on a

$$(\alpha_k = f(x^k)) \searrow \to \alpha, \beta_k \nearrow \to \beta, \alpha \geq \min f(S) \geq \beta.$$

Définition 3.3 *Une sélection est dite "complète" si pour chaque*

$$M \in \bigcup_{p=1}^{\infty} \bigcap_{k=p}^{\infty} L_k$$

on a

$$\inf(f(M \cap S)) \geq \alpha.$$

Une sélection est dite "borne-améliorante", si pour chaque itération, on choisit

$$M_{k,i} \in \arg\min\{\beta(M) : M \in L_k\}.$$

Théorème 3.4 (Horst[75]) *Supposons que S est fermé et que $\min f(S)$ existe. L'opération d'estimation de borne est supposée cohérente dans le prototype de SE. On a :*

1. *si la sélection est complète alors*

$$\alpha = \lim_{k\to\infty} \alpha_k = \lim_{k\to\infty} f(x^k) = \min f(S).$$

2. *si la sélection est borne-améliorante, alors*

$$\beta = \lim_{k\to\infty} \beta_k = \min f(S).$$

3. *si la sélection est complète, f est continue et $\{x \in S : f(x) \leq f(x^0)\}$ est borné, alors chaque point d'accumulation de $\{x^k\}$ résout le problème (P).*

Dans le Prototype de SE, il est nécessaire que $M \cap S \neq \emptyset$ pour chaque élément M de la partition. Par contre, dans des cas concrets, il n'y a pas de règles universelles pour décider de façon définitive si $M \cap S$ est vide ou non. En effet, pour définir un ensemble M, souvent un polyèdre, il suffit de connaître l'ensemble des sommets de M, noté $V(M)$. Mais évidemment, cela ne suffit pas à énoncer une décision concrète sur le caractère vide ou non de $M \cap S$, même dans un cas très simple. En conséquence, cette difficulté limite l'applicabilité du Prototype de SE. Dans la section suivante, on présente un deuxième prototype de SE qui n'est nécessaire pas de vérifier $M \cap S \neq \emptyset$ pour tous les ensembles M de la partition, mais seulement une partie suffisante d'eux permettant d'assurer la convergence de l'algorithme.

3.2 Séparation et Evaluation Progressive avec des ensembles non réalisables

Définition 3.5 *Un ensemble M est dit "non réalisable" si $M \cap S = \emptyset$. Un ensemble M est dit "réalisable" si $M \cap S \neq \emptyset$. Un ensemble M est dit "incertain" si on ne sait pas si M est réalisable ou non.*

Naturellement, un ensemble sera éliminé si on sait qu'il est non réalisable. Lorsque les ensembles incertains sont admis, on exige que

$$-\infty < \beta(M) \leq \min f(M \cap S), \text{ si M est réalisable};$$

$$-\infty < \beta(M) \leq \min f(M), \text{ si M est incertain.}$$

En général, $S_M \subset M \cap S$ peut être vide et il est possible que la borne $\alpha(M) = \infty$. La variante du prototype ci-dessous sera appliquée lorsqu'on ne peut pas décider définitivement si un élément $M \cap S$ de la partition donnée est réalisable ou non. Noter que dans ce cas, les bornes supérieures ne sont pas toujours disponibles. Nous allons ensuite décrire ce prototype modifié avec la convention que le minimum sur un ensemble vide prend la valeur infinie.

3.2.1 Prototype 2 de SE [76]

- Initialisation
  1. Choisir un compact $S \subset M_0$, un ensemble fini d'indices I_0, une partition $\mathcal{M}_0 = \{M_{0,i} : i \in I_0\}$ de M_0 satisfaisant $M_{0,i} \cap S \neq \emptyset, i \in I_0$.
  2. Pour chaque $i \in I_0$, déterminer

     $$S_{0,i} \subset M_{0,i} \cap S, S_{0,i} \neq \emptyset$$

     et la borne supérieure

     $$\alpha_{0,i} = \alpha(M_{0,i}) = \min f(S_{0,i}), x^{0,i} \in \arg\min f(S_{0,i}).$$

     Si $S_{0,i}$ n'est pas disponible (par des efforts raisonnables), on pose $S_{0,i} = \emptyset$.
  3. Pour chaque $i \in I_0$, déterminer la borne inférieure $\beta_{0,i} = \beta(M_{0,i})$ vérifiant

     $$\beta(M_{0,i}) \leq \min f(S \cap M_{0,i}), \textbf{ si M est réalisable}$$

     $$\beta(M_{0,i}) \leq \min f(S \cap M_{0,i}), \textbf{ si M est incertain.}$$

  4. Calculer

     $$\alpha_0 = \min_{i \in I_0} \alpha_{0,i}$$

     $$x^0 \in \arg\min f(x^{0,i}), i \in I_0$$

     $$\beta_0 = \min_{i \in I_0} \beta_{0,i}.$$

- Itération $k = 0, 1, \ldots$

Étape1 : Supprimer tout $M_{k,i} \in \mathcal{M}_k$ vérifiant

$$\beta_{k,i} \geq \alpha_k$$

où pour lequel on sait que $\min f(S)$ ne peut pas avoir lieu dans $M_{k,i}$. Soit L_k la liste des éléments restants $M_{k,i} \in \mathcal{M}_k$.
Si $L_k = \emptyset$ alors s'arrêter : x^k est une solution.

Étape2 : Sélectionner une partition $M_{k,i} \in L_k$, choisir un ensemble fini d'indices J_{k+1} et construire une partition pour $M_{k,i}$

$$\mathcal{M}_{k,i} = M_{k+1,i} : i \in J_{k+1}$$

de $M_{k+1,i}$ **en utilisant des règles pour éliminer les sous-ensembles non réalisables**.

Étape3 : Pour chaque $i \in J_{k+1}$ déterminer les bornes supérieures sur $M_{k+1,i}$,

$$S_{k+1,i} \subset M_{k+1,i} \cap S, S_{k+1,i} \neq \emptyset$$

et la borne supérieure

$$\alpha_{k+1,i} = \alpha(M_{k+1,i}) = \min f(S_{k+1,i}), x^{k+1,i} \in \arg\min f(S_{k+1,i}).$$

Si $S_{k+1,i}$ **n'est pas disponible, on pose** $S_{k+1,i} = \emptyset$.

Étape4 : Pour chaque $i \in J_{k+1}$, déterminer la borne inférieure sur $M_{k+1,i}$:

$$\beta_{k+1,i} = \beta(M_{k+1,i}) \leq \min f(S \cap M_{k+1,i}).$$

Étape5 : Poser

$$\mathcal{M}_{k+1} = (L_k \cap M_{k,i}) \cup \mathcal{M}_{k,i}$$

Soit I_{k+1} l'ensemble d'indices tels que

$$M_{k+1} = \{M_{k+1,i} : i \in I_{k+1}\}$$

est la partition actuelle.

Étape6 : Calculer

$$\alpha_{k+1} = \min_{i \in I_{k+1}} \alpha_{k+1,i}$$

$$x^{k+1} \in \arg\min f(x^{k+1,i}), i \in I_{k+1}$$

$$\beta_{k+1} = \min_{i \in I_{k+1}} \beta_{k+1,i}.$$

$k = k + 1$ et répéter l'Itération.

3.2.2 Conditions de convergence

Définition 3.6 *Soit* $\{M_{k_q}\}$ *une suite décroissante des ensembles générés par la procédure de division. Une procédure de division est dite "exhaustive" si la suite de diamètres* $d(M_{k_q})$ *associés à* M_{k_q} *vérifie*

$$\lim_{q \to \infty} d(M_{k_q}) = 0.$$

Évidemment, pour une suite décroissante des ensembles générés par une division exhaustive, on a

$$\lim_{q \to \infty} M_{k_q} = \bigcap_q M_{k_q} = \{\bar{x}\}$$

Définition 3.7 *L'opération d'estimation de la borne inférieure est qualifiée de "fortement cohérente" (strongly consistent) si pour n'importe quelle suite décroissante* $\{M_{k_q}\}$ *générée par une division exhaustive telle que*

$$M_{k_q} \longrightarrow \{\bar{x}\}, q \to \infty$$

il existe une sous-suite $\{M_{k'_q}\} \to f(\bar{x}), q \to \infty$

Définition 3.8 *L'élimination-par-non réalisabilité est appelée "certain à la limite" si pour n'importe quelle suite décroissante* $\{M_{k_q}\}$ *générée par une division exhaustive telle que* $M_{k_q} \longrightarrow \{\bar{x}\}, q \to \infty$, *on a*

$$\bar{x} \in S$$

On désigne par Y^α *l'ensemble des points d'accumulation de la suite* $\{y^k\}$ *de points correspondant à* β_k.

Soit $X^* = \arg\min f(S)$ l'ensemble des solutions optimales de (P). On a le théorème suivant :

Théorème 3.9 (Horst [76]) *Soit le Prototype 2 vérifiant les conditions suivantes :*

1. *la division est exhaustive ;*
2. *la sélection est borne-améliorante ;*
3. *la borne inférieure est fortement cohérente ;*
4. *l'élimination est certaine à la limite.*

Alors, on a

$$\beta := \lim \beta_k = \min f(S)$$

et

$$Y^\alpha \subset X^*.$$

3.3 Réalisation de SE

La réalisation d'un algorithme SE dépend du choix des opérations suivantes :

- Diviser $M_{k,i}$
- Sélectionner $M_{k,i}$ pour diviser
- Estimer les bornes

3.3.1 Stratégie de subdivision

Une idée naturelle pour la subdivision est de construire des éléments de partition $\mathcal{M}_k$ de formes simples de manière à pouvoir les manipuler facilement. En général, on utilise les polyèdres les plus simples comme les simplexes, les rectangles, les cônes, et les prismes.
Il faut également diviser ces polyèdres de telle manière que la procédure de subdivision soit exhaustive, ce qui est nécessaire pour assurer la convergence de la méthode SE.

1. Subdivision simpliciale

On construit des n-simplexes pour une subdivision de M_0. Il n'est pas difficile de construire le premier simplexe M_0 contenant S. Soit $M = conv\{v^0, \ldots, v^n\}$ un simplexe avec $n+1$ sommets $v^0, \ldots, v^n$ de dimension n. Alors, un point quelconque $s \in M$ peut être représenté par

$$s = \sum_{i=0}^{n} \lambda_i v^i, \sum_{i=0}^{n} \lambda_i = 1, \lambda_i \geq 0.$$

Soit $s \neq v^i, i = 0, \ldots, n$. Posons $J = \{j : \lambda_j > 0\}$. On peut construire un n-simplexe

$$M_j = conv\{v^0, \ldots, v^{j-1}, s, v^{j+1}, \ldots, v^n\}$$

en remplaçant un sommet v^j tel que $\lambda_j > 0$ par s. De cette manière, on peut construire une subdivision, appelée radiale, de M. Très souvent, on choisit s comme le milieu de la plus longue arête de M et l'on divise M en deux n-simplexes. Dans ce cas, on a une bissection classique de M. Il est facile de démontrer que la bissection est exhaustive. Pourtant, on constate que les procédures exhaustives de division ne sont pas très efficaces car la convergence de la méthode est assez lente. On suggère alors une stratégie adaptative où l'on choisit s comme un point obtenu dans le processus d'estimation de borne (par exemple s est le point correspondant à $\beta(M)$). Le problème de cette subdivision est qu'on ne peut plus assurer que la procédure de division est exhaustive. Certaines stratégies plus flexibles utiliser le plus souvent possible cette subdivision adaptative et à faire intervenir la bissection pour empêcher la dégradation. Une stratégie heuristique mais pratique a été proposée dans [76]. L'idée est de combiner la bissection standard et la bissection adaptative. Considérons un simplexe M et un point $w \in M$ qui s'écrit comme $w = \sum_{i=0}^{n} \lambda_i v^i$. Étant donné un nombre $\delta > 0$, si $\min\{\lambda_i : \lambda_i > 0, i = 0, \ldots, n\} > \delta$ alors on peut appliquer la bissection adaptative. Sinon on utilise la bissection standard. Les expériences indiquent que $\delta = \frac{1}{2n^2}$ est un choix convenable.

2. Subdivision rectangulaire

L'idée de la subdivision rectangulaire est de construire des n-rectangles pour M_0 et toute partie de subdivision.

$$M_0 = \prod_{i=1}^{n} [a_i, b_i]$$

doit être le rectangle le plus serré qui contient S. Si S est convexe, le rectangle M_0 peut être déterminé en résolvant $2n$ problèmes convexes

$$a_i = min\{x_i : x \in S\}, b_i = max\{x_i : x \in S\}, i = 1, \ldots, n.$$

Un rectangle M peut être divisé en deux rectangles ayant des volumes égaux via le milieu de la plus longue arête de M. Dans ce cas, on obtient la bissection standard de M. On peut vérifier aussi que pour une suite décroissante $\{M_k\}$ de rectangles générés par la bissection classique, il existe un point $\bar{x}$ tel que

$$\lim_{k\to\infty} M_k = \bigcap_k M_k = \{\bar{x}\}.$$

On suggère également d'utiliser une subdivision adaptative (comme dans le cas simplexe) que la bissection classique. Une bissection rectangulaire adaptative proposée par Muu [77] semble être plus efficace car la convergence n'exige pas le caractère exhaustif. Dans Horst et Tuy [49], un concept de subdivision rectangulaire normale (NRS) a été proposé pour la classe des problèmes de minimisation de fonction concave séparable.

3. Subdivision conique

L'idée de la subdivision conique est de construire des cônes centrés au point intérieur de S. Soit $w \in S$ un point intérieur de S. Étant donné M_0 un n-simplexe contenant S, M_0 a $n+1$ faces, notées $F_{0,i}, i = 1, \ldots, n+1$ de dimension $n-1$ qui sont des $(n-1)$-simplexes. Les cône polyédraux $C_{0,i}$ centrés en$w$ et engendrés par $F_{0,i}, i = 1, \ldots, n+1$ constituent une division de $\mathbb{R}^n$ avec l'ensemble de $n+1$ cônes. Évidemment, une division de face F en l'ensemble de simplexes $\{F_j\}$ aboutit à une division du cône C en l'ensemble de cônes $\{C_j\}$. En particulier, la bissection de cône est engendrée par la bissection de simplexes. Si la procédure de division de simplexes est exhaustive dans le sens où chaque suite décroissante de cônes $\{C_k\}$ (générée par cette procédure) tend vers un rayon sortant de w. La subdivision conique est très utile quand une solution globale se trouve sur la frontière de S. Le premier algorithme conique a été proposé par Tuy[78] pour la minimisation d'une fonction concave sur un polyèdre.

4. Subdivision prismatique

Cette subdivision est une extension de la subdivision conique quand le point w est considéré comme un point fictif à l'infini, ce qui permet de construire des prismes dans $\mathbb{R}^n \times \mathbb{R}$. Chaque simplexe ou rectangle M définit un prisme $T = \{(x,t) : x \in M\}$. Les procédures exhaustives de division de simplexes et de rectangles créent les procédures de division des prismes qui sont aussi exhaustives dans le sens où chaque suite décroissante des prismes $\{T_k\}$ (générée par cette procédure) tend vers une droite verticale.

3.3.2 Règle de sélection

Une manière plus simple pour la sélection est d'utiliser la "borne-améliorante" (lowest lower bound), c.à.d., on choisit à chaque itération de SE un ensemble

$$M_{k,i} \in \arg\min\{\beta(M) : M \in L_k\}$$

pour la division suivante. C'est une stratégie de sélection simple mais efficace, car un élément $M_{k,i}$ possédant la plus petite borne inférieure est probablement de contenir une solution globale. De plus, cette sélection améliore la plus petite borne inférieure à chaque itération. Pourtant, il existe d'autre règle de sélection. Par exemple :

(S1) On choisit $M_{k,i}$ le plus récent ensemble créé dans la liste L_k (deep first).

(S2) Soit $\delta(M)$ lié à la taille de M (e.g. le diamètre, le volume etc.). On peut choisir

$$M_{k,i} \in \arg\min\{\delta(M) : M \in L_k\}.$$

3.3.3 Estimation de la borne

Soit un ensemble M_k. On résout le problème $\beta(M_k) = \min f(M_k)$ pour estimer une borne inférieure. Si M_k et f sont convexes, c'est un problème de programmation convexe qui est facile à résoudre. En pratique, nous essayons souvent de construire un ensemble convexe T_k de forme simple (e.g. rectangle, simplexe etc.) et le plus petit que possible tel que $M_k \cap S \subset T_k \subset M_k\}$. La borne inférieure $\beta(M_k)$ va donc être remplacée par $\min f(T_k)$ qui est peut être plus proche de la valeur $\min f(M_k \cap S)$ que $\beta(M_k)$. $\min f(T_k)$ est appélé un problème relaxé du problème $\min f(M_k \cap S)$.

Pour estimer une borne supérieure, on peut évaluer le coût de la fonction objectif au point réalisable $x^* \in S$ et la valeur $f(x^*)$ est une borne supérieure. En pratique, pour obtenir une bonne borne supérieure, on utilise une approche locale pour trouver une solution réalisable. Dans nos travaux de recherches, nous utilisons toujours DCA comme un "solveur" de la borne supérieure pour résoudre le problème d'optimisation non convexe $\min f(M_k \cap S)$.

L'estimation de la borne constitue toujours un dilemme entre la convergence et l'efficacité. Un algorithme SE va converger plus rapidement si la borne supérieure et

la borne inférieure s'approchent l'une de l'autre plus rapidement. Pour accélérer la convergence, on construit souvent $T_k : M_k \cap S \subset T_k \subset M_k$ de manière à ce que la borne $\beta(M_k) = \min f(T_k)$ soit estimée en produisant des efforts raisonnables (e.g. un demi-espace). L'utilisation de T_k donne une certaine souplesse à l'estimation de borne. En particulier, elle permet de construire un plan de coupe. On peut combiner la technique de plan de coupe avec l'algorithme SE. Cette approche paraît prometteuse et montre sa supériorité par rapport à celles qui sont purement SE.

3.3.4 Technique de Relaxation DC

La technique de relaxation DC a pour but d'estimer une borne inférieure pour un programme DC. L'idée principale de la relaxation DC est de construire de manière simple un problème relaxé convexe dont la valeur optimale est une borne inférieure du programme DC étudié. La structure DC de la fonction objectif $f = g - h$ se prête bien à la recherche d'une minorante convexe de f sur un convexe fermé borné C pour la procédure d'évaluation dans SE. En effet, au lieu de calculer l'enveloppe convexe de f - ce qui est très difficile, voire impossible en général - on peut se contenter d'une bonne minorante convexe de f comme la somme de g et de l'enveloppe convexe de $-h$ sur C. Cela est réalisable, sachant que si C est un polyèdre convexe borné, l'enveloppe convexe d'une fonction concave est une fonction convexe polyédrale qui se calcule à l'aide d'un programme linéaire. De plus si C est un pavé (box) et h est séparable ou si C est un simplexe, cette enveloppe convexe devient une fonction affine explicite. En optimisation globale, une bonne estimation de borne inférieure se révèle cruciale car plus cette borne inférieure est proche de la fonction, plus la méthode SE sera efficace. Par conséquent, la technique de relaxation DC joue un rôle considérable en optimisation globale (notamment en combinant DCA avec SE).

3.3.4.1 Enveloppe convexe de fonction non convexe

Considérons le programme non convexe :

$$\alpha := \inf\{\psi(x) : x \in \mathbb{R}^n\}, \tag{3.1}$$

où $\psi : \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$ est une fonction propre ($\text{dom } \psi \neq \varnothing$) munie d'une minorante affine dans $\mathbb{R}^n$. La manière la plus optimale de formuler le problème (3.1) en un programme convexe est de construire l'enveloppe convexe de ψ par [47, 48] :

$$\text{co } \psi(x) := \inf\{\sum_i \lambda_i \psi(x_i) : \lambda_i \geq 0, \sum_i \lambda_i = 1, x \in \text{dom } \psi, x = \sum_i \lambda_i x_i\}, \tag{3.2}$$

où l'infimum est pris sur toutes les représentations de x comme combinaison convexe des éléments de x_i, de telle sorte que seul un nombre fini de λ_i est non nul. La fonction convexe co ψ avec

$$\text{dom co } \psi = \text{co dom } \psi \tag{3.3}$$

est la plus grande fonction convexe majorée par ψ. Elle conduit à des programmes convexes avec la même valeur optimale :

$$\alpha := \inf\{\text{co } \psi(x) : x \in \mathbb{R}^n\} = \inf\{\overline{\text{co}}\ \psi(x) : x \in \mathbb{R}^n\}. \tag{3.4}$$

Il est bien connu que [48] :
(i) $Arg\min\ \psi \subset Arg\min \text{ co } \psi \subset Arg\min\ \overline{\text{co}}\,\psi$
(ii) co $(Arg\min\ \psi) \subset \overline{\text{co}}\ (Arg\min\ \psi) \subset Arg\min\ \overline{\text{co}}\,\psi$
(iii) $\overline{\text{co}}\ \psi = \psi^{**}$.
(iv) Si ψ est une fonction s.c.i et 1-coercive (c.à.d., $\lim_{\|x\|\to+\infty} \frac{\psi(x)}{\|x\|} = +\infty$), alors co $\psi = \overline{\text{co}}\ \psi = \psi^{**}$.

Remarque 9 *(i) Le problème (3.1) peut être réécrit sous la forme :*

$$\alpha := \inf\{\psi(x) : x \in dom\ \psi\}, \tag{3.5}$$

cependant dans (3.4), on peut remplacer $x \in \mathbb{R}^n$ par $x \in$ co dom ψ. Comme d'habitude en analyse convexe, une fonction $\psi : C \subset \mathbb{R}^n \to \mathbb{R}$ est souvent identifiée à son extension $\psi + \chi_C$ pour tout $\mathbb{R}^n$.
(i) Soit $C \subset \mathbb{R}^n$ et $\psi : \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$ avec $C \subset dom\ \psi$, on note $co_C\ \psi$ l'enveloppe convexe de ψ sur C, c.à.d., $co_C\ \psi := co\ (\psi + \chi_C)$. De même, $\overline{co}_C\ \psi$ est synonyme de $\overline{co}\ (\psi + \chi_C)$.

Trouver l'enveloppe convexe d'une fonction non convexe est en général très difficile, sauf le cas des fonctions concaves sur des polyèdres convexes bornés (polytopes). On cherche plutôt des relaxations convexes plus faciles à manipuler pour calculer des bornes inférieures de la valeur optimale α, comme la relaxation DC est présentée dans la section suivante.

3.3.4.2 Enveloppe convexe de fonctions concaves sous ensemble polytope

Soit K un ensemble polytope non vide dont les sommets sont $V(K) := \{v^1, ..., v^m\}$. Alors $K =$ co $V(K)$. Les sommets $v^1, ..., v^m$ sont dits affinement indépendants s'il n'y a pas de nombres réels $\lambda_i, i = 1, ..., m$ non tous nuls tels que [47, 48]

$$\sum_{i=1}^{m} \lambda_i = 0 \text{ et } \sum_{i=1}^{m} \lambda_i v^i = 0. \tag{3.6}$$

Dans ce cas, K est appelé un $(m-1)-$ simplexe et tout $x \in K$ s'exprime de manière unique comme combinaison convexe de $v^1, ..., v^m$. Si ψ est une fonction concave finie sur K, alors l'expression (3.2) pour $\text{co}_K\ \psi$ devient plus simple et calculable ([49, 51, 22, 41])

Theorem 3.1 *Si ψ est une fonction concave finie sur K, on ait*

(i) $co_K\ \psi$ est la fonction polyédrale convexe sur K définie par :

$$co\ _K\ \psi(x) = \min\{\sum_{i=1}^{m} \lambda_i \psi(v^i) : \lambda_i \geq 0, \sum_{i=1}^{m} \lambda_i = 1, x = \sum_{i=1}^{m} \lambda_i v^i\}. \tag{3.7}$$

De plus, $co_K\ \psi$ et ψ coïncident sur $V(K)$.

(ii) Si K est un $(m-1)-$simplexe, alors $co_K\ \psi$ est la fonction affine définie par

$$co_K\ \psi(x) = \sum_{i=1}^{m} \lambda_i \psi(v^i), \lambda_i \geq 0, \sum_{i=1}^{m} \lambda_i = 1, x = \sum_{i=1}^{m} \lambda_i v^i. \tag{3.8}$$

3.3.4.3 Enveloppe convexe de fonction séparable

Soit $\psi = (\psi_1, ..., \psi_m)$ une fonction séparable sur $C = \Pi_{i=1}^m C_i$ avec $C_i \subset$ dom $\psi_i \subset \mathbb{R}^{n_i}$, $i = 1, ..., m$, c.à.d.,

$$\psi(x) = \sum_{i=1}^{m} \psi_i(x_i),\ \forall x = (x_1, ..., x_m) \in C, \tag{3.9}$$

alors $\mathrm{co}_C\ \psi$ peut être calculée explicitement à partir de $\mathrm{co}_{C_i}\ \psi_i$, $i = 1, ..., m$.

Proposition 3.10 *Si $\psi_i, i = 1, ..., m$ est minorée sur C_i par une fonction affine, alors pour chaque $x = (x_1, ..., x_m) \in K$*

$$co_C\psi(x) \geq \sum_{i=1}^{m} co_{C_i}\psi_i(x_i) \geq \sum_{i=1}^{m} \overline{co}_{C_i}\psi_i(x_i) = \sum_{i=1}^{m} (\psi_i + \chi_{C_i})^{**}(x_i) = \overline{co}_C\psi(x). \tag{3.10}$$

Preuve. Par hypothèse, $C \subset$ dom ψ et ψ est minorée par une fonction affine sur C donc, comme mentionné ci-dessus, $\overline{\mathrm{co}}_C\ \psi = (\psi + \chi_C)^{**}$ et $\overline{\mathrm{co}}_{C_i}\psi_i(x_i) = (\psi_i + \chi_{C_i})^{**}(x_i)$ pour $i = 1, .., m$ et $x_i \in C_i$. La première inégalité est triviale, car $\sum_{i=1}^m \mathrm{co}_{C_i}\psi_i(x_i)$ est une minorante convexe de ψ sur C. Pour la dernière égalité, il suffit de prouver que $\sum_{i=1}^m (\psi_i + \chi_{C_i})^{**}(x_i) = (\psi + \chi_C)^{**}(x)$ pour chaque $x = (x_1, ..., x_m) \in C = \Pi_{i=1}^m C_i$. Nous avons, pour $y = (y_1, ..., y_m) \in \Pi_{i=1}^m \mathbb{R}^{n_i}$,

$$\begin{aligned}
(\psi + \chi_C)^*(y_1, ..., y_m) &= \sup\{\langle x, y\rangle - \psi(x) : x \in C\} \\
&= \sup\{\sum_{i=1}^{m} [\langle x_i, y_i\rangle - \psi_i(x_i)] : x = (x_1, ..., x_m) \in C = \Pi_{i=1}^m C_i\} \\
&= \sum_{i=1}^{m} \sup\{[\langle x_i, y_i\rangle - \psi_i(x_i)] : x_i \in C_i\} = \sum_{i=1}^{m} (\psi_i + \chi_{C_i})^*(y_i).
\end{aligned}$$

Il s'ensuit que pour $x = (x_1, ..., x_m) \in \Pi_{i=1}^m \mathbb{R}^{n_i}$,

$$(\psi + \chi_C)^{**}(x) = \sum_{i=1}^{m} (\psi_i + \chi_{C_i})^{**}(x_i).$$

□

3.3.4.4 Minorante convexe de fonctions DC sur un ensemble convexe compact

Pour construire des minorantes convexes de fonctions DC en optimisation globale, nous considérons les programmes DC (3.11) avec la contrainte explicite C (un ensemble convexe non vide fermé et borné dans $\mathbb{R}^n$), $\varphi, \psi \in \Gamma_0(\mathbb{R}^n)$ telles que $C \subset \text{dom } \varphi \subset \text{dom } \psi$:

$$\alpha = \inf\{\theta(x) := \varphi(x) - \psi(x) \ : \ x \in C\}. \tag{3.11}$$

En conformité avec les résultats mentionnés dans 3.3.4.2 et 3.3.4.3, nous proposons la minorante calculable de θ sur C suivante :

(i) $\varphi + \text{co}_C(-h)$ si $V(C)$ est facile à calculer, par exemple, dans le cas où C est un ensemble convexe polyédral borné avec l'ensemble de sommets $V(C)$.

(ii) Pour le cas général, $\text{co}_C(-h)$ sera remplacé par

- $\overline{\text{co}}_L(-h)$ où L est un polytope contenant C défini dans 3.3.4.3, ($L_i := [a_i, b_i]$ est souvent utilisé en pratique), si h est séparable,

  ou

- $\text{co}_S(-h)$ avec S un simplexe contenant C.

La technique de relaxation DC a été utilisée et implémentée dans nos travaux de recherche (voir le chapitre 6).

Deuxième partie

Programmation mixte avec variables entières

CHAPITRE 4

Programmation quadratique convexe mixte avec variables entières

Sommaire

4.1 Introduction

Considering the optimization problem :

$$(MIQCP) \qquad \min \quad f(x,y) := x^T Q_0 x + y^T P_0 y + c_0^T x + d_0^T y$$

subject to :

$$(x,y) \in C$$

$$x \in \mathbb{R}^n, y \in \mathbb{Z}^m.$$

$C = P \cap Q$ is assumed to be a nonempty compact convex set, where $P := \{(x,y) \in \mathbb{R}^n \times \mathbb{R}^m : Ax + By \leq b, A_{eq}x + B_{eq}y = b_{eq}, x \in [\underline{x}, \bar{x}], y \in [\underline{y}, \bar{y}]\}$ is *a set of linear constraints* (including linear equalities, linear inequalities and box constraints) ; $Q := \{(x,y) \in \mathbb{R}^n \times \mathbb{R}^m : x^T Q_i x + y^T P_i y + c_i^T x + d_i^T y \leq s_i, i = 1, \ldots, L\}$ is *a set of convex quadratic constrains* (i.e., all matrices Q_i and P_i must be positive semi-definite). We will also assume that there is at least one coefficient of the variable y is not null. This kind of formulation is called a *Mixed Integer Quadratic Convex Program* (MIQCP). It is a nonconvex program since the present of integer variables will destroy the convexity of the convex constraint C.

Many classical combinatorial optimization problems are in fact one of the special cases of this formulation, such as the mixed 0-1 (resp. 0-1) linear program (MLP0-1) (resp. (LP0-1)), the mixed integer (resp. integer) linear program (MILP) (resp. (ILP)), the integer (resp. 0-1) quadratic convex program (IQCP) (resp (QCP0-1)). Consequently, there are a lot of applications in applied sciences, such as in Financial Optimization, Telecommunication, Data Mining, Bioinformation, Cryptography etc. Thus, solving this problem is very important although it is a well-known NP-Hard problem and difficult for global optimization. In the realm of DC programming, some papers have talked about solving the special case in which only binary variables are taken into account ([33, 41, 37]). However, there haven't any remarkable idea on problems with general integer variables for DC programming approaches.

The main difficulty lies in how to find an available DC formulation for handling the integer set. The well-known binary decomposition of integer variables is inefficient since huge a mount of additional variables must be introduced that involving large-scale mixed 0-1 programs. In this paper, more efficient *continuous representation methods* are investigated. This kind of techniques don't need any additional variable and thus suitable for handling large-scale problems. Moreover, this technique is very suitable for deriving DC programming formulation to (MIQCP). Then a very efficient DC programming approach (DCA) is applied for solving this problem. We also proposed a new hybride method combining DCA with B&B (DCA-BB) for the global optimization. During our research, we have firstly studied a special case on the general mixed integer linear program (MILP). The results have been published in 2008 [42]. In that paper, a hybrid method combining DCA with a branch and bound scheme, and linear relaxation is investigated that could be considered as a preliminary research on this topic. The paper could be found in the next chapter. Our researches in the current chapter will focus on the new developments for mixed integer convex quadratic programs via DC programming and DCA, B&B techniques, SDP relaxations etc. Some numerical simulations for the proposed approaches are also reported in the last section.

4.2 Representations of an integer set

The DC Algorithm (DCA) is a continuous optimization approach for solving DC program with convex constraint. In order to handle a discrete constraint, we must firstly find a continuous formulation for the discrete set. After that, we can transform the mixed integer problem as a DC program.

For instance, the binary set $\{0,1\}^n$ could be represented as $\{x \in [0,1]^n : p(x) \leq 0\}$, where the function $p : \mathbb{R}^n \to \mathbb{R}$ is continuous and verifying $p(x) \geq 0, \forall x \in [0,1]^n$, and $p(x) = 0$ if and only if $x \in \{0,1\}^n$. There are infinitely many functions p. In practice, we often use two classes of functions as follows :

1. $p(x) = \sum_{i=1}^{n} a_i x_i(1 - x_i)$, where $a_i > 0$.
2. $p(x) = \sum_{i=1}^{n} \min(\frac{b_i}{a_i} x_i, \frac{b_i}{a_i - 1}(x_i - 1))$, where $a_i \in]0, 1[, b_i > 0$.

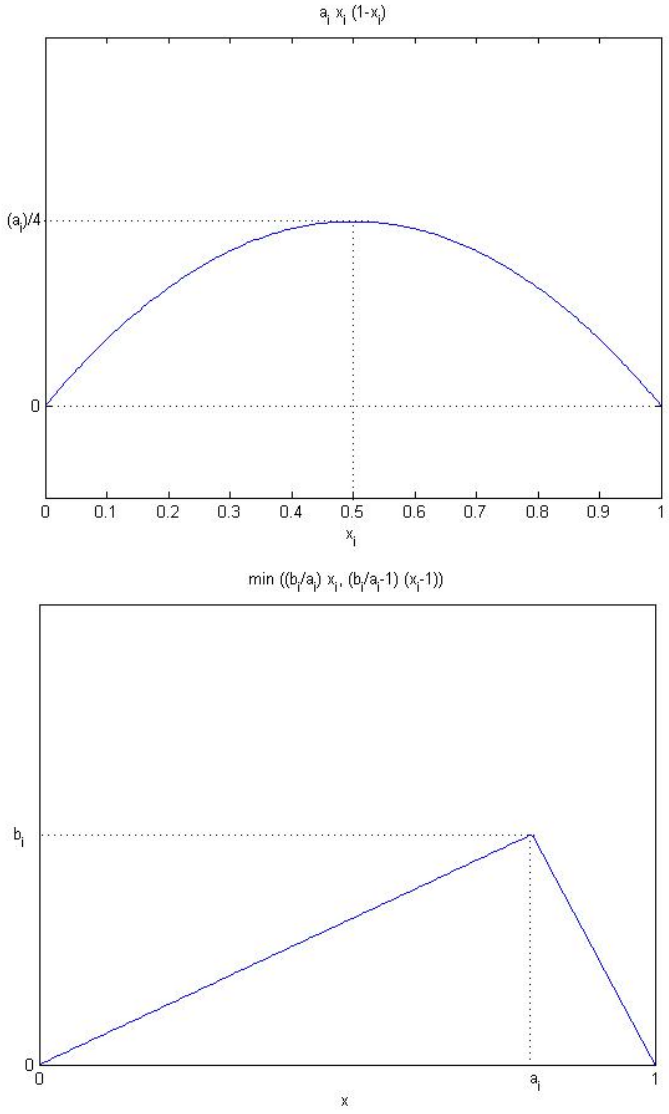


FIG. 4.1 – Functions $a_i x_i(1 - x_i)$ and $\min(\frac{b_i}{a_i} x_i, \frac{b_i}{a_i - 1}(x_i - 1))$

Note that these two classes both consist of *concave and separable* functions (see figure 4.1). More specifically, in the first class, the function $p(x) = \sum_{i=1}^{n} a_i x_i(1 - x_i)$ is a differentiable concave quadratic function ; while in the second class, $p(x) = \sum_{i=1}^{n} \min(\frac{b_i}{a_i} x_i, \frac{b_i}{a_i - 1}(x_i - 1))$ is constructed by the polyhedral concave functions that maybe nondifferentiable at some points. In practice, we quite often take $a_i = 1, i = 1, \ldots, n$ for the first class to obtain the function

$$p(x) = \sum_{i=1}^{n} x_i(1 - x_i) = x^T(1 - x)$$

and set $a_i = b_i = \frac{1}{2}, i = 1, \ldots, n$ in the second class to derive the function

$$p(x) = \sum_{i=1}^{n} \min(x_i, 1 - x_i).$$

Since p is concave, the set $\{p(x) \leq 0\}$ is a nonconvex set called *reverse convex set* or *anti-convex set*. The representation $\{x \in [0,1]^n : p(x) \leq 0\}$ is in fact the intersection of a box constraint (convex set) $[0,1]^n$ and a reverse convex constraint (nonconvex set) $\{p(x) \leq 0\}$.

The above representation is called "*continuous representation*" for a binary set. Thanks to the special proprieties of the function p, we can recur to the exact penalty techniques and/or SDP relaxation/reformulation techniques for rewriting the mixed integer program as a DC program.

Getting inspiration from the binary case, we consider the continuous representation for a general integer set. Some equivalent formulations are proposed in the form of $\{x : x \in \mathbb{R}^n, p(x) \leq 0\}$, with $p : \mathbb{R}^n \to \mathbb{R}^+$ being continuous function and verifying $p(x) = 0$ if and only if $x \in \mathbb{Z}^n$, and $p(x) > 0, \forall x \notin \mathbb{Z}^n$. Evidently, such functions p have infinitely many cases. For instance, we can use

$$p(x) := \sum_{i=1}^{n} (1 - \cos(2\pi x_i)). \tag{4.1}$$

where p is a separable function in $x_i, i = 1, \ldots, n$ (see figure 4.2) and holds the following properties :

Proposition 4.1 *Let* $p(x) = \sum_{i=1}^{n} (1 - \cos(2\pi x_i))$.

1) $0 \leq p(x) \leq 2n \ \forall x \in \mathbb{R}^n$;

2) $p(x) = 0$ *if and only if* $x_i \in \mathbb{Z}, i = 1, \ldots, n$;

3) $p(x) \in \mathbf{C}^\infty(\mathbb{R}^n, \mathbb{R}^+)$.

Another available function is

$$p(x) := \sum_{i=1}^{n} |\sin(\pi x_i)|. \tag{4.2}$$

It's also a separable function in x_i. The main difference between (4.1) and (4.2) is that the function p in (4.2) is nondifferentiable at all integer points (see figure 4.3), while p in (4.1) is a differentiable function.

The function p can be defined via the horizontal translation to integer units of the concave function $p(x) = x^T(1 - x)$ or $p(x) = \sum_{i=1}^{n} \min(x_i, 1 - x_i)$ as well, i.e., let

$$p_i(x_i) = \sup_{s_i \in \mathbb{Z}} \{(x_i - s_i)(1 - x_i + s_i)\}$$

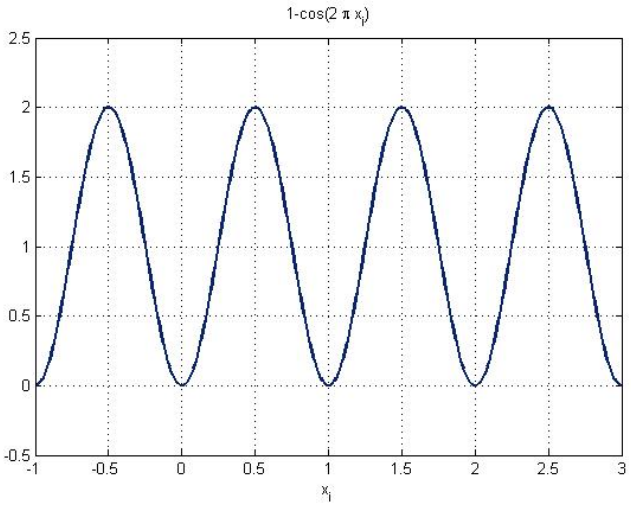


FIG. 4.2 – Function $1 - \cos(2\pi x_i)$

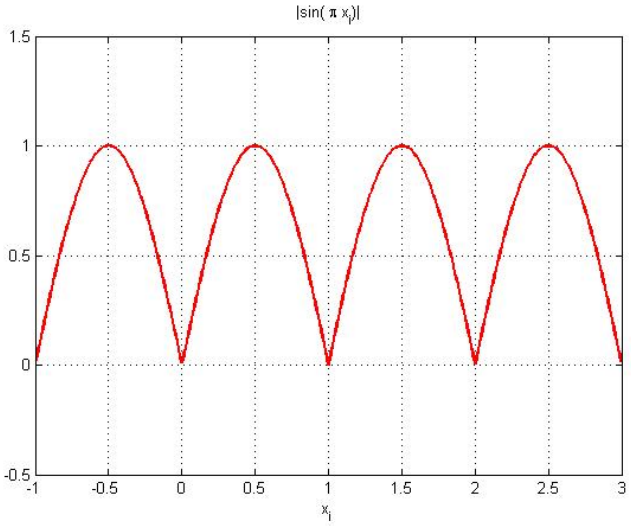


FIG. 4.3 – Function $|\sin(\pi x_i)|$

or

$$p_i(x_i) = \sup_{s_i \in \mathbb{Z}} \{\inf(x_i - s_i, 1 - x_i + s_i)\}$$

, the function p is

$$p(x) = \sum_{i=1}^{n} p_i(x_i).$$

The nature of the last class of functions is quite similar to $\sum_{i=1}^{n} |\sin(\pi x_i)|$ since they are both nondifferentiable at integer points, and locally concave on rectangles $[s, s+1], s \in \mathbb{Z}^n$. Particularly, when x_i is restricted in $[\underline{x}_i, \bar{x}_i]$ where $\underline{x}_i$ and $\bar{x}_i$ are supposed to be integer constants, then $\{x \in [\underline{x}, \bar{x}] : p(x) \leq 0\}$ represents the set of all

integers in the box $[\underline{x}, \bar{x}]$.

In virtue of the continuous representation techniques of the integer set, we can find a suitable function p to reformulate the problem (MIQCP) as a continuous formulation :

$$(P1) \qquad \min \quad f(x,y) := x^T Q_0 x + y^T P_0 y + c_0^T x + d_0^T y$$

subject to :

$$(x,y) \in C$$

$$p(y) \leq 0.$$

Obviously, the problem (MIQCP) is equivalent as (P1) which is a nonconvex program since the constraint $p(y) \leq 0$ is nonconvex. Without this constraint, the problem (P1) should be a convex quadratic program which can be efficiently solved by many existing methods and solvers, such as Interior Point Method and CPLEX solver.

For choosing a suitable function p, we can distinguish the binary case and the general integer case :

- If $y_i \in \{0,1\}$, then $p_i(y_i) = y_i(1-y_i)$ or $p_i(y_i) = \min(y_i, 1-y_i)$ is used.
- If y_i is a general integer variable, we suggest using $p_i(y_i) = 1 - \cos(2\pi y_i)$ or $p_i(y_i) = |\sin(\pi y_i)|$.

The function p is always given by $p(y) = \sum_{i=1}^{m} p_i(y_i)$.

In order to solve the problem (P1), we propose applying a robust and efficient local optimization algorithm for DC programming, called DCA, that can handle very well large-scale nonconvex programs (more details about DC programming and DCA can be found in the Chapter 1). For global optimization, we propose some new approaches combining DCA with a branch and bound scheme, as well as SDP relaxation techniques.

Note that the problem (P1) can not be handled directly by DCA due to the exist of the nonconvex constraint $p(y) \leq 0$. Fortunately, the problem can be reformulated as a DC program via penalization techniques, this topic will be discussed in the next section.

4.3 DC representations for (P1)

For establishing a DC formulation to the problem (P1), we can firstly construct its penalized problem with a penalty parameter t (a given positive constant) :

$$(P_t) \qquad \min\{F_t(x,y) := f(x,y) + tp(y) : (x,y) \in C\} \tag{4.3}$$

For all y in C must satisfy the equality $p(y) = 0$ at all integer points, and $p(y) > 0$ at all non integer points. Hence we can add a nonnegative penalty term $tp(y)$ with the function f to obtain the objective function F_t of the penalized problem. Minimizing F_t over the convex set C has an effect on punishing all non integer cases and enforcing y to be integer.

Note that (P_t) is a nonconvex optimization problem since the objective function F_t is nonconvex. For all functions p presented in the previous section, F_t can always be reformulated as a DC function :

For the binary case, since the functions $p_i(y_i) = y_i(1-y_i)$ and $p_i(y_i) = \min(y_i, 1-y_i)$ are both concave functions on C, they are already DC functions. Moreover, because sum of DC functions is also a DC function and f is a convex quadratic function on C, we have $f(x, y)+tp(y) = f(x, y)+t\sum_{i=1}^{m} p_i(y_i)$ is a DC function which has an obvious DC decomposition as :

$$F_t(x, y) = g(x, y) - h(y)$$

where $g(x, y) = f(x, y)$ and $h(y) = -tp(y)$ are both convex functions on C.

For the general integer case, it is easy to prove that there exists a suitable $\eta > 0$ such that the function $p_i(y_i) = 1 - \cos(2\pi y_i)$ has a DC decomposition as :

$$p_i(y_i) = [\frac{\eta}{2}y_i^2] - [\frac{\eta}{2}y_i^2 - p_i(y_i)]$$

Proposition 4.2 *There exists $\eta_0 > 0$ such that for all $\eta \geq \eta_0$, the function $p_i(y_i) = 1 - \cos(2\pi y_i)$ has a DC decomposition as*

$$p_i(y_i) = [\frac{\eta}{2}y_i^2] - [\frac{\eta}{2}y_i^2 - p_i(y_i)]$$

Proof. Since $p_i(y_i) = 1 - \cos(2\pi y_i)$ is a differentiable function, its first and second derivatives are

$$p_i'(y_i) = 2\pi \sin 2\pi y_i$$
$$p_i''(y_i) = 4\pi^2 \cos 2\pi y_i$$

Then

$$[\frac{\eta}{2}y_i^2 - p_i(y_i)]'' = \eta - p_i''(y_i) = \eta - 4\pi^2 \cos 2\pi y_i$$

It's clear that $\frac{\eta}{2}y_i^2$ is convex when $\eta > 0$. On the other hand, If $\frac{\eta}{2}y_i^2 - p_i(y_i)$ is convex then $[\frac{\eta}{2}y_i^2 - p_i(y_i)]'' \geq 0$, i.e., $\eta \geq 4\pi^2 \cos 2\pi y_i, \forall y_i \in \mathbb{R}$. We know that $4\pi^2 \cos 2\pi y_i \leq 4\pi^2$. So we can take $\eta_0 = 4\pi^2 > 0$, and then for all $\eta \geq \eta_0$, we have $\frac{\eta}{2}y_i^2$ and $\frac{\eta}{2}y_i^2 - p_i(y_i)$ are both convex functions, i.e.,

$$p_i(y_i) = [\frac{\eta}{2}y_i^2] - [\frac{\eta}{2}y_i^2 - p_i(y_i)]$$

is a DC decomposition for $\eta \geq 4\pi^2$. □

Finally, we give a DC decomposition of F_t as

$$\begin{aligned} F_t(x,y) &= [f(x,y) + t\sum_{i=1}^m \tfrac{\eta}{2} y_i^2] - t[\sum_{i=1}^m \tfrac{\eta}{2} y_i^2 - p_i(y_i)] \\ &= g(x,y) - h(y) \end{aligned}$$

where $g(x,y) = f(x,y) + \frac{t\eta}{2}\|y\|^2$ is convex quadratic function on C and $h(y) = t[\frac{\eta}{2}\|y\|^2 - p(y)]$ (with $p(y) = \sum_{i=1}^m 1 - \cos(2\pi y_i)$) is a convex function on C.

For the nondifferentiable cases, the functions $p_i(y_i) = |\sin(\pi y_i)|$, $p_i(y_i) = \sup_{s_i\in\mathbb{Z}}[(y_i - s_i)(1-y_i+s_i)]$ and $p_i(y_i) = \sup_{s_i\in\mathbb{Z}} min(y_i - s_i, 1-y_i+s_i)$ are all DC functions. The reason is that $p^i_{s_i}(y_i) = (y_i - s_i)(1-y_i+s_i)$ and $p^i_{s_i}(y_i) = min(y_i - s_i, 1-y_i+s_i)$ are both concave functions (i.e., DC functions) on $\mathbb{R}$. Therefore, the function $p_i(y_i) = \sup_{s_i\in\mathbb{Z}} p^i_{s_i}(y_i)$ is also a DC function. Particularly, the integer $s_i \in [\underline{y}_i, \bar{y}_i] \cap \mathbb{Z}$ is finite since y_i is restricted in the set $[\underline{y}_i, \bar{y}_i]$. For a given integer $i = 1, \ldots, m$, we can represent the function p_i as a DC function

$$p_i(y_i) = \sup_{s_i\in[\underline{y}_i,\bar{y}_i]\cap\mathbb{Z}} p^i_{s_i}(y_i) = \sum_{s_i\in[\underline{y}_i,\bar{y}_i]\cap\mathbb{Z}} p^i_{s_i}(y_i) - \inf_{s\in[\underline{y}_i,\bar{y}_i]\cap\mathbb{Z}}\{\sum_{s_i\in[\underline{y}_i,\bar{y}_i]\cap\mathbb{Z}, s_i\neq s} p^i_{s_i}(y_i)\}$$

For the case where $p_i(y_i) = |\sin(\pi y_i)|$, we can represent $|\sin(\pi y_i)|$ as

$$|\sin(\pi y_i)| = \sup_{s\in\mathbb{Z}} \tilde{p}^i_s(y_i)$$

where $\tilde{p}^i_s(y_i)$ is a piecewise function on $\mathbb{R}$ defined by

$$\tilde{p}^i_s(y_i) = \begin{cases} sin(\pi y_i), & y_i \in [s, s+1]\,; \\ \pi(y_i - s), & y_i < s\,; \\ -\pi(y_i - s - 1), & y_i > s+1. \end{cases}$$

$\tilde{p}^i_s(y_i)$ is a continuous and differentiable concave function on $\mathbb{R}$, therefore $|\sin(\pi y_i)| = sup_{s\in\mathbb{Z}}(\tilde{p}^i_s)$ is also a DC function.

The DC decomposition of the above three nondifferentiable functions are difficult to be applied in numerical computation because of their complicated formulation. Therefore, in practice, we suggest using the simple and elegant DC decomposition of the differentiable function $p_i(y_i) = 1 - \cos(2\pi y_i)$ for handling the general integer case.

Since $p_i(y_i) = 1 - \cos(2\pi y_i)$ is suitable for both binary and integer cases. Therefore, we have two different ways to construct the function p. One is using $p_i(y_i) = 1 - \cos(2\pi y_i)$ only, the other one is combine the binary and integer cases to construct p as follows :
Let $I = \{i \in \{1, 2, \ldots, m\} : y_i \in \{0, 1\}\}$ be the index set for binary variables. the function p is defined as :

$$p(y) = \sum_{i\in I} p^1_i(y_i) + \sum_{i\notin I} p^2_i(y_i) \tag{4.4}$$

where

$$p_i^1(y_i) = y_i(1 - y_i) \text{ or } p_i^1(y_i) = \min(y_i, 1 - y_i)$$

and

$$p_i^2(y_i) = 1 - \cos(2\pi y_i).$$

Now, we give DC programming formulations for the penalized problem (P_t) as

$$(PDC_t) \qquad \min\{g_t(x, y) - h_t(y) : (x, y) \in C\} \tag{4.5}$$

where the convex functions g_t and h_t are defined as one of the following formulas :

1.
$$g_t(x, y) = f(x, y) + \frac{t\eta}{2} \sum_{i \notin I} y_i^2$$

and

$$h_t(y) = t[\sum_{i \notin I} (\frac{\eta}{2} y_i^2 - p_i^2(y_i)) - \sum_{i \in I} p_i^1(y_i)].$$

2.
$$g_t(x, y) = f(x, y) + \frac{t\eta}{2} \|y\|^2$$

and

$$h_t(y) = t[\frac{\eta}{2} \|y\|^2 - \sum_{i=1}^{m} p_i(y_i)].$$

4.4 Penalty techniques in nonlinear mixed integer programming

In the previous section, we constructed the penalized problem as well as its DC programming formulation for a mixed integer nonlinear program. We will focus in this section the relationships between the solution set of the penalized problem and the original problem.

Considering a general nonlinear mixed integer program :

$$\begin{array}{ll} \min & f(x) \\ s.t & c_i(x) = 0, i = 1, \ldots, m_1, \\ & c_j(x) \geq 0, i = m_1 + 1, \ldots, m, \\ & \underline{x}_i \leq x_i \leq \bar{x}_i, i = 1, \ldots, n, \\ & x_i \in \mathbb{Z}, i = n_1, \ldots, n. \end{array} \tag{4.6}$$

Let us denote

$$S_0 = \{x \in \mathbb{R}^n : c_i(x) = 0, i = 1, \ldots, m_1; c_j(x) \geq 0, i = m_1 + 1, \ldots, m; \underline{x}_i \leq x_i \leq \bar{x}_i, i = 1, \ldots, n\};$$

$$S = \{x \in S_0 : x_i \in \mathbb{Z}, i = n_1, \ldots, n\}.$$

We make assumptions that :

- S_0 and S are both nonempty sets ;
- f is twice continuously differentiable functions in $\mathbb{R}^n$, and $c_i, i = 1, \ldots, m$ are continuous functions in $\mathbb{R}^n$;
- there exist constants L_1 and L_2 such that

$$\|\nabla f(x)\|_1 \leq L_1 \text{ and } \|\nabla^2 f(x)\|_1 \leq L_2, \forall x \in S_0.$$

Definition 4.3 *x is an integer point if $x \in \mathbb{Z}^n$. x is a quasi-integer point if $x_i \in \mathbb{Z}, i = n_1, \ldots, n$.*

Definition 4.4 *The ε - cubic neighborhood of a quasi-integer point $\tilde{x}$ is defined as the set*

$$\mathrm{N}_\varepsilon(\tilde{\mathrm{x}}) = \{\mathrm{x} : ||\mathrm{x} - \tilde{\mathrm{x}}||_\infty \leq \varepsilon, \mathrm{x_i} = \tilde{\mathrm{x}}_\mathrm{i}, \mathrm{i} = 1, \ldots, \mathrm{n_1}\}.$$

Every feasible solution in S is a quasi-integer point, so a global optimal solution of the mixed integer program (4.6) is the best quasi-integer point in S for minimizing the function f.

We consider the penalized problem of (4.6) defined by

$$\begin{array}{ll} \min & f(x) + tp(x) \\ s.t & x \in S_0 \end{array} \tag{4.7}$$

where $p(x) = \sum_{i=n_1}^{n}(1 - cos(2\pi x_i))$, and $t > 0$ is a given large positive number.

It is not difficult to understand that if $t > 0$ and large enough, then a global minimizer x^* of the penalized problem (4.7) should verify that $tp(x^*)$ is very close to zero (see [79]), and then $p(x^*)$ must be close to zero since t is a large positive number. That is to say x^* is closed to a quasi-integer point of S_0. Under the assumption that S is nonempty, there is at least one quasi-integer point $\hat{x} \in S$ satisfying $p(\hat{x}) = 0$ and $f(\hat{x}) \approx f(x^*) + tp(x^*)$. Therefore, if t is large enough, we can obtain at least an approximate optimal solution of (4.6) by globally solving its penalized problem (4.7).

It is clear that when $\varepsilon \geq \frac{1}{2}$ then ε - cubic neighborhoods of all quasi-integer points cover the whole space $\mathbb{R}^n$, since for any point $x \in \mathbb{R}^n$, there exists at least one quasi-integer point $\hat{x}$ such that $\|x - \hat{x}\|_\infty \leq \frac{1}{2} \leq \varepsilon \Longleftrightarrow x \in \mathrm{N}_{\frac{1}{2}}(\hat{\mathrm{x}}) \subset \mathrm{N}_\varepsilon(\hat{\mathrm{x}})$.

Now, let us consider the case $\varepsilon < \frac{1}{2}$. We can generalize the theorems proposed in papers [80, 81] and replace $\frac{1}{5}$ by ε to derive the following Theorems 4.5, 4.6 and 4.7 :

Theorem 4.5 (See [80, 81]) *For a given $0 < \varepsilon < \frac{1}{2}$, there exists a t (depends on ε) large enough such that*

1. *Every quasi-integer point $\tilde{x} \in S$ whose $N_\varepsilon(\tilde{x})$ includes at least one local optimal solution of the problem (4.7).*
2. *Every local optimal solution of the penalized problem (4.7) could be in a ε - cubic neighborhood of a quasi-integer point $\tilde{x} \in S$.*

The theorem 4.5 means that for a given $\varepsilon > 0$, there exists a t (depends on ε) such that every minimizer of the problem (4.7) (defined with t) is included in a ε - cubic neighborhood of certain quasi-integer point of S. An important feature of this theorem is that : we can reduce the value of $\varepsilon > 0$ as small as you wish, and if ε is small enough then there exists a t large enough such that only one quasi-integer point $\hat{x}$ whose ε - cubic neighborhood $N_\varepsilon(\hat{x})$ includes a local optimal solution of (4.7). This also means that for such appropriate ε and t, a local optimal solution of (4.7) will be included in only one $N_\varepsilon(\hat{x})$.

Theorem 4.6 (See [80, 81]) *For a given $\varepsilon > 0$, there exists a $t > 0$ large enough such that a global minimizer of the penalized problem (4.7) defined with t is included in a ε - cubic neighborhood of a global quasi-integer solution $\tilde{x}$ of the problem (4.6).*

Corollary 4.7 (See [80, 81]) *For a given $\varepsilon > 0$, if x^* is a global minimizer of the penalized problem (4.7) and included in a ε - cubic neighborhood of a global quasi-integer solution $\tilde{x}$ of the problem (4.6), then t is large enough.*

The above theorems 4.5, 4.6 and 4.7 show that for a large enough t, we can solve the penalized problem (4.7) to obtain an approximate quasi-integer solution of the initial mixed integer program. Furthermore, if the global optimal solution of the penalized problem is in a ε-cubic neighborhood of a feasible quasi-integer point, then it is also a solution of the mixed integer program.

Obviously, t depends on ε. The smaller the ε is, the bigger the t should be. However, it is well-known that when t is too big, the penalized problem becomes ill-conditioned, and difficult to be numerically solved. So choosing an appropriate ε (resp. t) is important. In practice, we often take ε in an interval $[0.1, 0.5]$. For instance, $\frac{1}{5}$-cubic neighborhood is suggested in the papers [80, 81], and the estimation of t is also proposed in their works. We can generalize the estimation formulation of t by replacing $\frac{1}{5}$ as $\varepsilon > 0$.

4.5 DC Algorithms for solving the penalized problem

In the previous two sections, we have explained that a mixed integer program can be solved by tackling its penalized problem, and the penalized problem can be reformulated as DC programs.

Let's consider the first DC programming formulation :

$$(PDC1_t) \qquad \min\{g(x,y) - h(y) : (x,y) \in C\} \tag{4.8}$$

where $C = \{(x,y) \in \mathbb{R}^n \times \mathbb{R}^m : Ax + By \leq b, A_{eq}x + B_{eq}y = b_{eq}, x \in [\underline{x}, \bar{x}], y \in [\underline{y}, \bar{y}], x^T Q_i x + y^T P_i y + c_i^T x + d_i^T y \leq s_i, i = 1, \ldots, L\}$ is assumed to be a nonempty compact convex set.

$$g(x,y) = f(x) + \frac{t\eta}{2}\sum_{i \notin I} y_i^2$$

and

$$h(y) = t[\sum_{i \notin I}(\frac{\eta}{2}y_i^2 - p_i^2(y_i)) - \sum_{i \in I} p_i^1(y_i)]$$

are both convex functions on C, where $f(x,y) := x^T Q_0 x + y^T P_0 y + c_0^T x + d_0^T y$ is a convex quadratic function on C. $I = \{i \in \{1,2,\ldots,m\} : y_i \in \{0,1\}\}$, $p_i^1(y_i) = y_i(1-y_i)$ or $p_i^1(y_i) = \min(y_i, 1-y_i)$ and $p_i^2(y_i) = 1 - \cos(2\pi y_i)$.

For solving this DC program, the general framework of DCA yields constructions of two sequences $\{X^k := (x^k, y^k)\}$ and $\{Y^k := (u^k, v^k)\}$ defined as :

$$\begin{array}{ccc} (x^k, y^k) & \longrightarrow & (u^k, v^k) \in \partial h(x^k, y^k) \\ & \swarrow & \\ (x^{k+1}, y^{k+1}) \in \partial g^*(u^k, v^k) & \longrightarrow & (u^{k+1}, v^{k+1}) \in \partial h(x^{k+1}, y^{k+1}). \end{array}$$

For computing $\partial h(x^k, y^k)$, we need considering the differentiability of the function h. If h is differentiable at the point (x^k, y^k), then $\partial h(x^k, y^k) = \{\nabla h(x^k, y^k)\}$. Otherwise, the subdifferential $\partial h(x^k, y^k) = \{(u,v) \in \mathbb{R}^n \times \mathbb{R}^m : h(x,y) \geq h(x^k, y^k) + \langle (x,y) - (x^k, y^k), (u,v)\rangle, \forall (x,y) \in \mathbb{R}^n \times \mathbb{R}^m\}$ is a closed convex set.

In our studied problem, $h(x,y) = h(y) = t[\sum_{i \notin I}(\frac{\eta}{2}y_i^2 - p_i^2(y_i)) - \sum_{i \in I} p_i^1(y_i)]$. if the index set I is nonempty then h is nondifferentiable at some points of C since $p_i^1(y_i) = \min(y_i, 1-y_i), \forall i \in I$ is nondifferentiable at $y_i = 0.5$; Otherwise, h is differentiable on $\mathbb{R}^n \times \mathbb{R}^m$. Fortunately, It is easy to compute the subdifferential for both differentiable and nondifferentiable cases. We have the following results :

$$\partial p_i^1(x,y) = \{\nabla p_i^1(x,y)\} = \{(1-2y_i)e^{i+n}\} \tag{4.9}$$

$$\partial p_i^1(x,y) = \{we^{i+n}\}, w = \begin{cases} 1, & y_i < 0.5\,; \\ \text{-1}, & y_i > 0.5\,; \\ \text{[-1,1]}, & y_i = 0.5. \end{cases} \tag{4.10}$$

$$\partial p_i^2(x,y) = \{\nabla p_i^2(x,y)\} = \{2\pi sin(2\pi y_i)e^{i+n}\} \tag{4.11}$$

where e^{i+n} is the $(i+n)$-th unit vector of $\mathbb{R}^{n+m}$.

Therefore, the $\partial h(x,y)$ could be explicitly computed by

$$\partial h(x,y) = \{t[\sum_{i\notin I}(\eta y_i e^{i+n} - \nabla p_i^2(x,y)) - \sum_{i\in I}\partial p_i^1(x,y)]\} \tag{4.12}$$

It follows from (4.9), (4.10), (4.11) and (4.12) that

$$(u^k,v^k) \in \partial h(x^k,y^k) \Leftrightarrow \begin{cases} u^k = 0, \\ v^k = [t\sum_{i\notin I}(\eta y_i^k - 2\pi sin(2\pi y_i^k))e^i] - t\sum_{i\in I}\partial p_i^1(y^k) \end{cases} \tag{4.13}$$

where e^i is the i-th unit vector of $\mathbb{R}^m$ and

$$\partial p_i^1(y^k) = \{(1-2y_i^k)e^i\}$$

or

$$\partial p_i^1(y^k) = \{we^i\}, w = \begin{cases} 1, & y_i^k < 0.5\,; \\ \text{-1}, & y_i^k > 0.5\,; \\ \text{[-1,1]}, & y_i^k = 0.5. \end{cases}$$

For computing $X^{k+1} = (x^{k+1},y^{k+1}) \in \partial g^*(u^k,v^k)$, it is equivalent to solve the following convex quadratic program :

$$(x^{k+1},y^{k+1}) \in \arg\min\{g(x,y) - \langle (x,y),(u^k,v^k)\rangle : (x,y)\in C\}. \tag{4.14}$$

where $g(x,y) = x^TQ_0x + y^TP_0y + c_0^Tx + d_0^Ty + \frac{t\eta}{2}\sum_{i\notin I} y_i^2$ is a convex quadratic function on C.

This DC algorithm could be terminated when one of the following conditions satisfied :

1. One of the sequences $\{X^k\}$ or $\{Y^k\}$ converge, i.e.,

$$\|X^{k+1} - X^k\| \le \varepsilon_1 \text{ or } \|Y^{k+1} - Y^k\| \le \varepsilon_1.$$

2. The sequence $\{F_t(X^k) = g(x^k,y^k) - h(y^k)\}$ converges, i.e.,

$$\|F_t(X^{k+1}) - F_t(X^k)\| \le \varepsilon_2.$$

3. If (x^k,y^k) is in a ε_3-cubic neighborhood of a quasi-integer point $(x^k,\bar{y}^k)$.

The first and the second stopping criterion are general convergence conditions for DCA. The third one is due to the theorems 4.5 and 4.6 that a minimizer must be in a ε_3-cubic neighborhood of a quasi-integer point. This condition will accelerate the convergence of DCA. In practice, we will take $\varepsilon_3 = \frac{1}{5}$ for searching solutions in $\frac{1}{5}$-cubic neighborhood.

Let's describe the first DC algorithm (DCA1) as follows :

DC Algorithm 1 (DCA1)

Initialization :
Choose an initial point $X^0 = (x^0, y^0) \in \mathbb{R}^n \times \mathbb{R}^m$.
t is a fixed large positive number.
Let ε_1, ε_2 be some small positive numbers, and $\varepsilon_3 = \frac{1}{5}$.
Let iteration number $k = 0$.

Iteration :
- Calculate $(u^k, v^k) \in \partial h(x^k, y^k)$ via the explicit formula (4.12).
- Solve the quadratic convex program (4.14) to obtain (x^{k+1}, y^{k+1}).

Check stopping conditions :
If (x^k, y^k) is in a $\frac{1}{5}$ - cubic neighborhood of a quasi-integer point $(x^k, \tilde{y})$ and this point is a feasible solution in C
Then **STOP** algorithm and $(x^k, \tilde{y})$ is a computed solution.

If either $||X^k - X^{k-1}|| \leq \varepsilon_1$
or $|F_t(X^k) - F_t(X^{k-1})| \leq \varepsilon_2$
Then **STOP** algorithm and (x^k, y^k) is a computed solution.
Otherwise, $k = k + 1$ and repeat **Iteration**.

Theorem 4.8 (Convergence properties of DCA1) *The algorithm DCA1 has the following convergence properties :*

1. *DCA1 will generate convergence sequences $\{(x^k, y^k)\}$ and $\{F_t(x^k, y^k)\}$, and the sequence $\{F_t(x^k, y^k)\}$ is decreasing and bounded below.*
2. *DCA1 converges either to a feasible quasi-integer solution (an upper bound solution), or a Karush-Kuhn-Tucker point of the problem $(PDC1_t)$.*

The proof of the convergence of (DCA1) is obvious based on the general convergence theorem of the DCA (see [29, 23, 32]).

Let us consider the second DC programming formulation with only $p_i(y_i) = 1 - \cos(2\pi y_i)$ for both binary and integer cases. The DC program is given by

$$(PDC2_t) \qquad \min\{g(x, y) - h(y) : (x, y) \in C\} \tag{4.15}$$

where

$$g(x, y) = f(x) + \frac{t\eta}{2}\|y\|^2$$

and

$$h(y) = t[\frac{\eta}{2}\|y\|^2 - \sum_{i=1}^{m} p_i(y_i)].$$

In this case, the computation of ∂h is more concise as

$$(u^k, v^k) \in \partial h(x^k, y^k) \Leftrightarrow (u^k = 0, v^k = t\eta y^k - 2t\pi \sin 2\pi y^k). \tag{4.16}$$

For computing $X^{k+1} = (x^{k+1}, y^{k+1}) \in \partial g^*(u^k, v^k)$, we can solve the following convex quadratic program :

$$(x^{k+1}, y^{k+1}) \in \arg\min\{g(x, y) - \langle (x, y), (u^k, v^k)\rangle : (x, y) \in \mathbf{C}\}. \tag{4.17}$$

where $g(x, y) = x^T Q_0 x + y^T P_0 y + c_0^T x + d_0^T y + \frac{t\eta}{2}\|y\|^2$ is a convex quadratic function on C. The stopping conditions are the same as in the (DCA1). We have the second DC Algorithm (DCA2) described as follows :

DC Algorithm 2 (DCA2)

Initialization :
Choose an initial point $X^0 = (x^0, y^0) \in \mathbb{R}^n \times \mathbb{R}^m$.
t is a fixed large positive number.
Let ε_1, ε_2 be some small positive numbers, and $\varepsilon_3 = \frac{1}{5}$
Let iteration number $k = 0$.

Iteration :
- Calculate $(u^k, v^k) \in \partial h(x^k, y^k)$ via the explicit formula (4.16).
- Solve the quadratic convex program (4.17) to obtain (x^{k+1}, y^{k+1}).

Check stopping conditions :
If (x^k, y^k) is in a $\frac{1}{5}$ - cubic neighborhood of a quasi-integer point $(x^k, \tilde{y})$ and this point is a feasible solution in C
Then **STOP** algorithm and $(x^k, \tilde{y})$ is a computed solution.

If either $||X^k - X^{k-1}|| \leq \varepsilon_1$
or $|F_t(X^k) - F_t(X^{k-1})| \leq \varepsilon_2$
Then **STOP** algorithm and (x^k, y^k) is a computed solution.
Otherwise, $k = k + 1$ and repeat **Iteration**.

The convergence of (DCA2) is also obvious due to the general convergence theorem of the DC algorithm whose proof can be found in the references ([29, 23, 32]).

Theorem 4.9 (Convergence properties of DCA2) *The algorithm DCA2 has the following convergence properties :*

1. *DCA2 will generate convergence sequences $\{(x^k, y^k)\}$ and $\{F_t(x^k, y^k)\}$, and the sequence $\{F_t(x^k, y^k)\}$ is decreasing and bounded below.*
2. *DCA2 converges either to a feasible quasi-integer solution (an upper bound solution), or a Karush-Kuhn-Tucker point of the problem $(PDC2_t)$.*

4.6 Initial point strategies for DCA

The DCA algorithms proposed in the above section need choosing an initial point $X^0 = (x^0, y^0) \in \mathbb{R}^n \times \mathbb{R}^m$. In general, the choice of a "good" initial point is important for DCA since it will affect the convergence rate of DCA, as well as the quality of the numerical solution. However, it is hard to answer the questions : "What is a "good" initial point ? And how to find it ?" We can give a definition on what kind of points should be "good" for DCA.

Definition 4.10 *For a given DC program and its DCA, a good initial point means that staring from which, DCA can converge to a global optimal solution of the DC program.*

We known that if its optimal solution exists, then the set of good initial points must be not empty, since the solution itself is a good initial point (It can be easily verified that DCA starting from an optimal solution will stop immediately at this point).

In the set of all good initial points denoted by G, we say a point $A \in G$ is better than another point $B \in G$ if and only if starting DCA from A needs less iteration than from B to converge to optimal solutions. Therefore, all optimal solutions are also best initial points in G. We also know that, if an optimal solution is an interior point of the constraint set, then there exists a neighborhood of this solution included in G. The set G is possible very large, for instance, we know that DCA can solve any convex program (both for constrained and unconstrained problems), For these cases, G is identical to the whole space $\mathbb{R}^n$. This example also shows that, a good initial point is not necessarily feasible to constrained DC program.

For general nonconvex programming, we haven't precise method neither for finding a good initial point, nor for determining whether a given point is good or not. Finding a good initial point is still an open question.

In practice, we often solve a relaxation problem to estimate an initial point. The quality of the estimated initial point will be checked according to the performance of the DCA (such as the number of iterations and the quality of the computed solution obtained by DCA). For our specific problem, if most of the integer variables are binaries, we can use the SDP relaxation techniques to binary variables.

Let $I = \{i \in \{1, 2, \ldots, m\} : y_i \in \{0, 1\}\}$, since the binary set $\{0, 1\}$ is exactly the solution set of the equation $y_i^2 = y_i$, then we can rewrite the binary constraints $y_i \in \{0, 1\}, i \in I$ as the quadratic equalities $y_i^2 = y_i, i \in I$.
Let us introduce an additional matrix variable $W = yy^T$, it follows that $y_i^2 = W_{ii}$ and we have the following equivalence

$$\begin{aligned} y_i \in \{0,1\}, i \in I &\iff y_i^2 = y_i, i \in I \\ &\iff \begin{cases} W = yy^T \\ W_{ii} = y_i, i \in I. \end{cases} \\ &\iff \begin{cases} \begin{bmatrix} 1 & y^T \\ y & W \end{bmatrix} \succeq O, \\ Tr(W) - \|y\|^2 \leq 0, \\ W_{ii} = y_i, i \in I. \end{cases} \end{aligned}$$

In the last formulation, the only nonconvex constraint is $Tr(W) - \|y\|^2 \leq 0$ which is in fact a reverse convex constraint (anti-convex constraint). We can relax this constraint as a linear one as :

$$Tr(W) + \langle \alpha, y \rangle + \beta \leq 0$$

where α and β are defined as

$$\alpha = -(\underline{y} + \bar{y})$$
$$\beta = \langle \underline{y}, \bar{y} \rangle$$

Since y is restricted in a box $[\underline{y}, \bar{y}]$, the following inequality holds :

$$Tr(W) - (\underline{y} + \bar{y})^T y + \langle \underline{y}, \bar{y} \rangle \leq Tr(W) - \|y\|^2, \forall y \in [\underline{y}, \bar{y}].$$

Now, we have a SDP relaxation problem as :

$$(RQP1) \qquad \min \quad f(x,y) := x^T Q_0 x + y^T P_0 y + c_0^T x + d_0^T y$$

subject to :

$$(x,y) \in C$$
$$\begin{bmatrix} 1 & y^T \\ y & W \end{bmatrix} \succeq O,$$
$$W_{ii} = y_i, i \in I,$$
$$Tr(W) - (\underline{y} + \bar{y})^T y + \langle \underline{y}, \bar{y} \rangle \leq 0.$$

The SDP relaxation problem is a convex program. Many SDP solvers such as Yalmip, PENNON can efficiently solve this problem.

Note that (RQP1) can be established if and only if there is binary variables. Otherwise, we have the general relaxation techniques for integer programming via considering all integer variables y as continuous variables. Thus we have the following quadratic convex relaxation problem :

$$(RQP2) \qquad \min \quad f(x,y) := x^T Q_0 x + y^T P_0 y + c_0^T x + d_0^T y$$

subject to :

$$(x,y) \in C$$

The (RQP2) is a Convex Quadratically Constrained Quadratic Program that can be very efficiently solved by existing solvers such as CPLEX, Yalmip etc.

Note that (RQP2) could be solved more efficiently than (RQP1), especially for large-scale problems. Since (RQP1) has less variables and constraints than (RQP1). However (RQP1) perhaps provide tighter convex relaxation and better lower bound. Therefore, it's worthwhile to compare these two relaxation methods in computation experiments.

4.7 Global Optimization Approaches GOA-DCA

DCA is a local optimization technique. In order to evaluate the quality of the computed solution provided by DCA and for developing global optimization method, we propose combining DCA with an adaptive Branch-and-Bound (B&B) scheme : GOA-DCA.

The B&B framework of GOA-DCA consists of two basic operations : Branching and Bounding.

4.7.0.5 Branching

The branching process aims at partitioning the convex set C into some subsets, then we can solve the optimization problem over these subsets in order to localize one subset containing the global optimal solution. The feasible set of the mixed-integer program is in fact the set of all quasi-integer points of C. Instead of partitioning the set C, we can perform the subdivision with the integer variable y.

Let us denote $M_0 = C$. Suppose that we have applied DCA for solving the DC program

$$\min\{F_t(x, y) = g(x, y) - h(y) : (x, y) \in M_0\}. \tag{4.18}$$

and found a computed solution (x^*, y^*). If it is a feasible quasi-integer solution, then it is considered as the current upper bound ; Otherwise, one of the following cases must be held :

1. The closest quasi-integer point of this solution is outside of M_0.
2. DCA converges to a KKT point which is not a local minimizer of the penalized problem.

We can find an index $i \in \{1, \ldots, m\}$ such that $y_i^* \notin \mathbb{Z}$. The suggested method is choosing the index i as :

$$i = \arg\max\{|\bar{y}_i - \underline{y}_i| : y_i^* \notin \mathbb{Z}\}.$$

Then, an adaptive rectangular subdivision is proposed as follows :

Let $\lfloor y_i^* \rfloor$ (resp. $\lceil y_i^* \rceil$) denotes the floor (resp. ceil) number of y_i^*. We can add linear constraints $y_i \leq \lfloor y_i^* \rfloor$ and $y_i \geq \lceil y_i^* \rceil$ respectively to derive the following two subdivision problems :

$$\min\{F_t(x,y) = g(x,y) - h(y) : (x,y) \in M_1^1\}. \tag{4.19}$$

$$\min\{F_t(x,y) = g(x,y) - h(y) : (x,y) \in M_1^2\}. \tag{4.20}$$

where

$$M_1^1 = M_0 \cap \{y_i \leq \lfloor y_i^* \rfloor\}$$

and

$$M_1^2 = M_0 \cap \{y_i \geq \lceil y_i^* \rceil\}$$

Let S denote the feasible set of the original mixed integer program. We have $S \subset M_0$, $M_1^1 \cap M_1^2 = \emptyset$ and $M_1^1 \cup M_1^2 \supset S$.

This kind of subdivisions must be terminated at finitely many times since the variable y is bounded in C thus the number of total integer points y in C is also finite.

4.7.0.6 Bounding

The bounding process consists of evaluating the lower and upper bounds. If we can solve the optimization problems as (4.19) and (4.20) in all partition sets and choose the best one between them then it is equivalent to solve the initial mixed integer program. However, global solving these subproblems is not a easy task since they are nonconvex DC programs. Instead of solving the them, we suggest establishing and solving their lower bound problems on the partition sets M_1^1 and M_1^2 defined as :

$$\beta(M_1^1) = \min\{f(x,y) : (x,y) \in M_1^1\}. \tag{4.21}$$

$$\beta(M_1^2) = \min\{f(x,y) : (x,y) \in M_1^2\}. \tag{4.22}$$

The lower bound problems (4.21) and (4.22) are established by replacing the nonconvex objective function $F_t(x,y) := f(x,y) + tp(y)$ by the convex quadratic function $f(x,y)$, and the inequality $F_t(x,y) = f(x,y) + tp(y) \geq f(x,y)$ holds for all $(x,y) \in C$. Therefore, the problems (4.21) and (4.22) are both convex quadratic programs, and can be efficient solved. Their optimal values $\beta(M_1^1)$ and $\beta(M_1^2)$ are lower bounds on these partition sets.

Restarting DCA

After solving a lower bound problem, we can evaluate the objective value of the nonconvex function $F_t(x,y)$ at the lower bound solution in order to check if we have found a promising point for restarting DCA. If the value of the DC function at this point is small than the current best upper bound then DCA is restarted from this point. A better feasible quasi-integer point could be found for updating the current best upper bound.

4.7.0.7 GOA - DCA for solving MIQCP

We combine DCA and B&B for establishing a new global optimization method called GOA-DCA. In this algorithm, the B&B framework forms a binary search tree in a way that we always create new branches from a node possessing a smallest lower bound (a node is correspond to a subdivision problem). The root of this tree is the initial DC program. During the computational process of B&B, we can prune away all nodes whose lower bounds are greater than the current best upper bound as well as the nodes whose solution set is empty.

GOA-DCA terminates when all nodes of the search tree were pruned away or when the gap between the current best upper bound and the smallest lower bound of the current existing nodes is less than a given tolerance.

Now, we can describe the proposed GOA-DCA as :

Global Optimization Algorithm GOA - DCA

Initialization :
Define ε as a sufficiently small positive number.
Solving the SDP relaxation problem (RQP1) or QCQP relaxation problem (RQP2) to find an initial point (x^0, y^0).
IF $(x^0, y^0) \in \mathrm{N}_{\frac{1}{5}}(\mathrm{x}^0, \tilde{\mathrm{y}}^0)$, and $(x^0, \tilde{y}^0)$ is a quasi-integer point in C
THEN we terminate algorithm, and $(x^0, \tilde{y}^0)$ is the computed solution.
ELSE let iteration number $k = 0$, $M_0 = C$, $\mathcal{S} = \{M_0\}$,
$LB = \{f(x^0, y^0)\}$ and $UB = +\infty$
ENDIF
Goto Step1.

Step1 :
Delete all subsets in S whose lower bound are greater or equal to UB.

IF $S = \emptyset$ or $|UB - \min(LB)| < \varepsilon$
THEN algorithm is terminated, the upper bound solution corresponding to UB is the computed solution.
ENDIF
Choosing a set $M_k \in S$ who has a smallest lower bound in LB.
Let $S = S \setminus M_k$
IF $F_t(x^k, y^k) \leq UB$ **THEN** we start DCA1 or DCA2 with the initial point (x^k, y^k) to find a computed solution (x^*, y^*).
ENDIF

IF $(x^*, y^*) \in \mathrm{N}_{\frac{1}{5}}(\mathrm{x}^*, \tilde{\mathrm{y}}^*)$, and $(x^*, \tilde{y}^*)$ is a quasi-integer point in C.

THEN
IF $F_t(x^*, \tilde{y}^*) < UB$, **THEN** $UB = F_t(x^*, \tilde{y}^*)$. **ENDIF**
ENDIF
Goto Step2.

Step2 :
• $i = \arg\max\{|\bar{y}_i - \underline{y}_i| : y_i^* \notin \mathbb{Z}\}$.
• Subdivide the set M_k into two subsets :
$M_{k,1} = M_k \cap \{y_i \leq \lfloor y_i^* \rfloor\}$ and $M_{k,2} = M_k \cap \{y_i \geq \lceil y_i^* \rceil\}$. Goto Step3.

Step3 :
Construct the lower bound problems (4.21) and (4.22) on $M_{k,1}$ and $M_{k,2}$.
Solving them to obtain the lower bound solutions $(x^{k,1}, y^{k,1})$ and $(x^{k,2}, y^{k,2})$
and their lower bounds $\beta(M_{k,1})$ and $\beta(M_{k,2})$.
IF (4.21) is a feasible problem and $\beta(M_{k,1}) < UB$
THEN Restart DCA with initial point $(x^{k,1}, y^{k,1})$ to obtain
a computed solution $(\bar{x}^{k,1}, \bar{y}^{k,1})$.
IF $(\bar{x}^{k,1}, \bar{y}^{k,1}) \in \mathrm{N}_{\frac{1}{5}}(\bar{\mathrm{x}}^{\mathrm{k},1}, \tilde{\mathrm{y}}^{\mathrm{k},1})$, and $(\bar{x}^{k,1}, \tilde{y}^{k,1})$ is a quasi-integer point in C
THEN $UB = F_t(\bar{x}^{k,1}, \tilde{y}^{k,1})$
ENDIF
Let $S = S \cup M_{k,1}$, $LB = LB \cup \{\beta(M_{k,1})\}$.
EDNIF
ENDIF
(The same operations to the problem (4.22) on the set $M_{k,2}$.)
Let $k = k + 1$ goto Step1.

4.8 Computational Experiments

We implement our algorithm DCA1, DCA2 and GOA-DCA in MATLAB R2008 with YALMIP interface and the CPLEX solvers for Linear and QCQP subproblems, as well as SDPA, SeDuMi et PENBMI for SDP relaxation problems. Numerical simulations are realized on a laptop equipped with Windows Vista SP2, Intel Core2 Duo CPU P8400 2.26GHz, 4G RAM, 32-bit system.

The test data are randomly generated in a way that : the positive semi-definite matrices for the convex quadratic functions are constructed by the MATLAB routine "sprandsym". Other matrices are generated by the function "randn". For generating tests with binary variables, we restrict the variable in [0,1].

The first tests will focus on the comparisons between the two relaxation problems (RQP1) and (RQP2). We want to find a cheap and high quality relaxation to provide an initial point for DCA. The quality of a relaxation problem depends on the quality

of its optimal solution and its computational time. Its optimal objective value is in fact a lower bound of the DC program. Therefore, the lower bound is greater, the quality is better. Here, the SDP problem (RQP1) is solved by SDPA solver, and the QCQP problem (RQP2) is solved by CPLEX. The following Table 4.4 shows some comparison results :

TAB. 4.4 – (RQP1) vs. (RQP2)

Prob	n	m	#0-1	#LC	#QC	RQP1		RQP2	
						obj	time(sec)	obj	time(sec)
P1	10	10	5	14	0	115.8386	0.7311	115.8386	0.7612
P2	20	15	10	8	2	229.8095	0.4122	229.8095	0.0899
P3	30	20	20	11	1	238.6122	0.5839	238.6122	0.1355
P4	30	30	20	26	1	1510.212	3.2111	1510.212	0.1793
P5	50	50	30	28	0	395.1381	118.62	395.1381	0.2182

In Table 4.4, the second column n is the number of continuous variables, and column m is the number of integer variables. The fourth column $\#0-1$ is the number of binary variables. #LC (resp. #QC) is the number of linear (resp. quadratic) constraints. We can observe that (RQP2) can almost always find the same optimal solution as (RQP1) and the solution time for (RQP1) is much more expensive than (RQP2) especial for large-scale problems. Therefore, the (QCQP) relaxation is a better choice for providing an initial point for DCA. This example told us that, the SDP relaxation is not always better than the standard convex relaxation even if it seems more efficient for the mixed $0-1$ program stated in many papers. From now on, we use QCQP relaxation for the following tests.

Our purpose in the next test is comparing the performances (computational time, quality of solutions) between (DCA1) and (DCA2). The penalized parameters $t = 10000$ and $\eta = 4\pi^2$ are fixed. We start (DCA1) and (DCA2) with the same initial point computed by QCQP relaxation. Some numerical results are presented in the table 4.5 :

In Table 4.5, the column "#C" denotes the total number of linear and quadratic constraints. The column "LB" is the lower bound at the initial point provided by the QCQP relaxation problem. We can observe that the (DCA2) can always converge to a feasible quasi-integer solution whose objective value is also close to the lower bound, i.e., the gap between the computed solution of DCA with the lower bound is often relatively small. This is a good result because the smaller the gap is, the better the quality of the computed solution should be. On the other hand, (DCA2) is better than (DCA1) for the tested problems, since (DCA2) always obtains feasible solution, while (DCA1) finds infeasible one for some of tests. Concerning the computation time, we haven't found obvious difference between these two methods. (DCA2) sometimes converges within less iteration than (DCA1) especially for relatively large-scale

TAB. 4.5 – (DCA1) vs. (DCA2)

Pb	n	m	#0-1	#C	LB	DCA1				DCA2			
						obj	#iter	time	feas	obj	#iter	time	feas
P1	10	10	5	20	1386.633	17266.21	1	0.0733	T	1386.736	2	0.1518	T
P2	10	10	10	30	1419.082	1419.143	10	1.0427	F	1419.446	3	0.3070	T
P3	30	20	20	15	869.1316	871.3163	9	0.9087	F	873.4826	6	0.6223	T
P4	30	30	20	23	38911.67	39563.38	8	1.2828	T	39563.38	8	1.2321	T
P5	40	30	20	22	70906.09	71134.44	5	0.9053	F	71719.52	8	1.5346	T
P6	50	50	30	12	60552.35	60553.59	4	1.5269	T	60553.59	4	1.4971	T
P7	60	50	20	33	42596.51	42660.50	8	2.8356	T	42660.50	6	2.0822	T
P8	80	60	10	32	15646.06	15710.38	8	3.3661	T	15710.38	8	3.3128	T
P9	90	80	80	130	628.2688	628.2688	1	0.9350	T	628.2688	1	0.9350	T
P10	200	100	50	200	82846.34	82873.11	11	19.294	F	82878.78	8	15.998	T

problem.

In the next table 4.6, we will present some comparison results between DCA2, GOA-DCA and CPLEX 11. The CPLEX is one of the most efficient and powerful linear, convex quadratic and mixed integer commercial solver nowadays. We compare with its Mixed Integer Optimizer "mipopt" with default parameters. The objective of the tests is to observe the quality of the solution obtained by DCA2 comparing with the global optimal solution given by GOA-DCA and CPLEX.

TAB. 4.6 – DCA2 vs. GOA-DCA vs. CPLEX

Pb	n	m	#0-1	#C	DCA2				GOA-DCA			CPLEX		
					obj	#iter	time	feas	obj	time	feas	obj	time	feas
P1	10	10	5	8	202.8921	3	0.5682	T	202.8921	1.81824	T	202.8921	2.3795	T
P2	10	10	10	10	186.2482	4	2.0611	T	186.2482	6.59552	T	186.2482	2.3257	T
P3	30	20	20	15	76.28531	6	2.3236	T	76.28531	7.43552	T	76.28531	4.0626	T
P4	30	30	20	22	28876.32	7	3.2157	T	27972.38	10.2902	T	27972.38	8.3557	T
P5	40	30	20	25	14983.02	5	3.3566	T	14982.90	10.7411	T	14982.90	6.5668	T
P6	50	50	30	5	7459.131	2	0.4396	T	7459.131	1.40672	T	7459.131	0.8058	T
P7	60	50	20	10	77332.86	5	1.3856	T	77332.86	4.43392	T	77332.86	2.6559	T
P8	80	60	10	23	26888.32	7	1.5715	T	26888.32	5.02881	T	26888.32	3.6408	T
P9	90	80	80	50	182563.9	2	1.1738	T	182563.9	3.75616	T	182563.9	4.3748	T
P10	200	100	90	100	76073.82	10	13.236	T	76073.56	32.3552	T	76073.56	22.302	T

We are delighted to see in the table 4.6 that DCA2 often provides global optimal solution. Moreover, the total iterations of DCA2 is often less than 10 iterations and its total computational time is very cheap. Comparing the computational time and the optimal solution between the proposed GOA-DCA and CPLEX, we found that GOA-DCA always obtains the same optimal solution as CPLEX, and CPLEX is slightly faster than GOA-DCA. We must notice that our algorithms are implemented in Matlab platform, while CPLEX is coded in C++. It is very hopeful that the C/C++ implementation of GOA-DCA will get faster than CPLEX.

4.9 Conclusion

In this paper, we propose new efficient local and global DC programming approaches for solving the Mixed Integer Convex Quadratic Convex Program (MIQCP). We firstly studied some continuous reformulation techniques for integer constraints and their specific properties as well as the continuous programming formulations for (MIQCP) via penalty techniques. Then we developed the appropriate DC decompositions and their related DCAs. We introduced a new stopping criterion to DCA for accelerating its convergence. The initialization strategy for DCA is also investigated. Moreover, a new hybrid global optimization method GOA-DCA combining DCA with a branch and bound scheme is proposed. Some numerical simulations comparing DCA, GOA-DCA, and CPLEX show that starting with a good initial point, DCA often converges quickly to a feasible quasi-integer solution, and a global ones in almost all cases. GOA-DCA is as fast as CPLEX for globally solving (MIQCP). DCA and GOA-DCA are high-quality and super-performance local and global optimization algorithms for Mixed Integer Quadratic Convex Program.

Concerning Mixed Integer Quadratic Nonconvex Program, it is crucial to develop efficient DC decomposition of the general quadratic programming problem, especially for handling large-scale problems. Some preliminary researches are reported in the chapters of polynomial programming.

Chapitre 5

Programmation linéaire mixte avec variables entières

An Efficient DC Programming Approach for Mixed-Integer Linear Program *

Yi-Shuai NIU[1], Tao PHAM DINH[2], and Hoai An LE THI[3]

Laboratoire de Mathématiques de l'INSA,
National Institute for Applied Sciences - Rouen,
BP 08, Place Emile Blondel F 76131, Mont Saint Aignan Cedex, France.
[1]niuys@insa-rouen.fr,[2]pham@insa-rouen.fr.
Laboratory of Theoretical and Applied Computer Science (LITA EA 3097)
UFR MIM, University of Paul Verlaine - Metz, Ile du Saulcy, 57045 Metz, France.
[3]lethi@uni-metz.fr

Abstract. In this paper, we propose a new efficient algorithm for globally solving a class of Mixed-Integer Program (MIP). If the objective function is linear with both continuous variables and integer variables, then the problem is called a Mixed-Integer Linear Program (MILP). Researches on MILP are important in both theoretical and practical aspects. Our approach for solving a general MILP is based on DC Programming & DCA (DC Algorithm). Using a suitable penalty parameter, we can reformulate MILP as a DC program which could be solved by DCA (a very efficient local optimization algorithm for DC programming). Furthermore, a robust global optimization algorithm (GOA-DCA): A hybrid method which combines DCA with a suitable Branch-and-Bound (B&B) method for globally solving general MILP problem is investigated. Moreover, the purposed method is also applicable to the Integer Linear Program (ILP). An illustrative example and some computational results show the robustness, the efficiency and the globality of our algorithm.

Key words: MIP, MILP, ILP, DC Programming, DCA, Branch-and-Bound, GOA-DCA

1 Introduction

The MIP problems are classical discrete optimization problems with both integer and continuous variables. Considering a general formulation of MILP:

$$\begin{array}{ll} \min & f(x,y) := c^T x + d^T y \\ s.t. & Ax + By \leq b, A_{eq}x + B_{eq}y = b_{eq}, \\ & (lb_x, lb_y) \leq (x,y) \leq (ub_x, ub_y), \\ & x \in \mathbb{R}^n, y \in \mathbb{Z}_+^m. \end{array} \tag{1}$$

* Published on CCIS 14, MCO 2008, pp. 244-253.

where the objective function $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ is a linear function. The variables x (resp. y) are bounded by lb_x and ub_x (resp. lb_y and ub_y) which are constants specifying the lower and upper bounds of x (resp. of y). The difficulty of (1) lies in the variable y, because it is an integer variable which destroys the convexity of the constraint set. If we suppose y is a continuous variable, then (1) becomes to a linear program which can be solved efficiently.

There are several well-known methods for solving MIPs: Branch-and-Bound Method, Cutting-Plane Method, Decomposition Method [7], etc. Some of them have already been implemented in commercial software, such as "ILOG CPLEX", "MATLAB", "XPressMP", "AIMMS", "LINDO/LINGO", "1stOpt", etc. Moreover, there are also open-source codes [11], such as "OSI","CBC" for MILP, and "BONMIN", the latter is a good solver for general Mixed Integer Nonlinear Program (MINLP). Although, we already have several methods and softwares for solving MIPs, it should be emphasized that the research on MIP should be continued, because MIP is classified as NP-hard, for which there is no efficient polynomial time algorithm. The methods mentioned above are often very expensive in computation time. Therefore, there is a need to improve these methods or to find more efficient methods.

In order to overcome the difficulty of the integer variables, we will reformulate the problem (1) to a DC program, then we apply an efficient method for solving DC programming, called DC Algorithm (DCA), which enable us to find a KKT point of the DC reformulation problem. DCA can rapidly generate a convergent sequence on which the objective values decrease.

In order to check the globality of the computed solution obtained by DCA and to guarantee that we can globally solve MILP, we combine DCA with a suitable Branch-and-Bound scheme (GOA-DCA). Some numerical results to the applications of DCA and GOA-DCA for (1) are also reported in the final section of the paper.

2 DC Reformulation for MILP

2.1 Reformulation of Integer Set

In this section, we will reformulate the integer set $\{y : y \in \mathbb{Z}_+^m\}$ using a twice continuously differentiable function.
Let

$$p(y) := \sum_{i=1}^{m} (1 - \cos(2\pi y_i)), \tag{2}$$

where $y \in \mathbb{R}^m$. The function $p : \mathbb{R}^m \to \mathbb{R}$ has some interesting properties which can be verified without any difficulty:

1) $0 \leq p(y) \leq 2m \ \forall y \in \mathbb{R}^m$;

2) $p(y) = 0$ if and only if $y_i \in \mathbb{Z},$ for all $i = 1, \ldots, m$;
3) $p(y) \in \mathbf{C}^\infty(\mathbb{R}^m, \mathbb{R})$.

By virtue of the third property, we can calculate the first and the second derivatives of $p(y)$, the gradient and the Hessian matrix, as follows

$$\nabla_y p(y) = 2\pi \begin{bmatrix} \sin 2\pi y_1 \\ \cdots \\ \sin 2\pi y_m \end{bmatrix} = 2\pi \sin 2\pi y, \tag{3}$$

$$\nabla_y^2 p(y) = 4\pi^2 \mathrm{Diag}(\cos 2\pi \mathrm{y}_1, \cdots, \cos 2\pi \mathrm{y}_\mathrm{m}), \tag{4}$$

where $\mathrm{Diag}(\cos 2\pi y_1, \cdots, \cos 2\pi y_m)$ is a $m \times m$ diagonal matrix.

Therefore, the spectral radius of $\nabla_y^2 p(y)$, denoted by $\rho(\nabla_y^2 p(y))$, satisfies the following inequality:

$$\rho(\nabla_y^2 p(y)) \ := \ \max_{1 \le i \le m} (4\pi^2 |\cos 2\pi y_i|) \le 4\pi^2. \tag{5}$$

With the help of the properties 1) and 2), the set $\{y : y \in \mathbb{Z}_+^p\}$ can be represented as

$$\{y : y \in \mathbb{Z}_+^p\} \ \equiv \ \{y : p(y) = 0, y \in \mathbb{R}_+^p\} \equiv \ \{y : p(y) \le 0, y \in \mathbb{R}_+^p\}. \tag{6}$$

2.2 Reformulation of MILP as a DC program

Let $\mathbf{K} := \{(x, y) \in \mathbb{R}^n \times \mathbb{R}_+^m : Ax + Gy \le b, A_{eq}x + B_{eq}y = b_{eq}, lb_x \le x \le ub_x, lb_y \le y \le ub_y\}$ be a nonempty and compact set.

The problem MILP (1) can be expressed as

$$\min\{f(x, y) := c^T x + d^T y : (x, y) \in \mathbf{K}, y \in \mathbb{Z}_+^m\}. \tag{7}$$

Using (6), we reformulate the problem (7) as an equivalent problem:

$$\min\{f(x, y) : (x, y) \in \mathbf{K}, p(y) \le 0\}. \tag{8}$$

We establish a penalized problem for (8) with a penalty parameter t (a positive constant):

$$\min\{F_t(x, y) := c^T x + d^T y + tp(y) : (x, y) \in \mathbf{K}\}. \tag{9}$$

Note that (9) is a nonlinear and nonconvex optimization problem. The additional term $tp(y)$ in the objective function satisfies the inequality $tp(y) \ge 0$ for all $y \in \mathbb{R}^m$, and it is equal to 0 if and only if $y \in \mathbb{Z}^m$. According to the general result of the penalty method (see [8]), for a given large number t, the minimizer of (9) should be found in a region where p is relatively small.

Definition 1. (See [6].) Let $\tilde{y} \in \mathbb{Z}^m$. The set $\mathrm{N}(\tilde{y}) = \{\mathrm{y} : ||\mathrm{y} - \tilde{\mathrm{y}}||_\infty \leq \frac{1}{5}\}$ is called a $\frac{1}{5}$ - *cubic neighborhood* of the integer point $\tilde{y}$.

Theorem 1. *(See [6].)Suppose that t is large enough, if (x^*, y^*) is a global minimizer of (9) and y^* is in a $\frac{1}{5}$ - cubic neighborhood of an integer point $\tilde{y}$, then $(x^*, \tilde{y})$ is a solution of the problem (7).*

However, the problem (9) is a difficult nonlinear optimization problem. Fortunately, we can represent p as a DC function:

$$p(y) = (\frac{\eta}{2} y^T y) - (\frac{\eta}{2} y^T y - p(y)), \tag{10}$$

where $\eta \geq \rho(\nabla_y^2 p(y))$. Note that $(\frac{\eta}{2} y^T y) - (\frac{\eta}{2} y^T y - p(y))$ is a DC function (difference of two convex functions) if $\eta \geq \rho(\nabla_y^2 p(y))$. The reason is that the functions $\frac{\eta}{2} y^T y$ and $\frac{\eta}{2} y^T y - p(y)$ are convex, because their Hessian matrices are semi-positive definite matrices when the inequality $\eta \geq \rho(\nabla_y^2 p(y))$ satisfies. Using (5) in the section 2.1, $\rho(\nabla_y^2 p(y)) \leq 4\pi^2$, we can take $\eta = 4\pi^2$ to establish an available DC decomposition of p :

$$p(y) = 2\pi^2 y^T y - (2\pi^2 y^T y - p(y)).$$

Thus, the problem (9) can be reformulated as a DC program:

$$\min\{F_t(x, y) := g(y) - h(x, y) : (x, y) \in \mathbf{K}\}. \tag{11}$$

where $g(y) := 2t\pi^2 y^T y$ is a convex quadratic function,
$h(x, y) := (2t\pi^2 y^T y - tp(y) - d^T y) - c^T x$ is a separable convex function.

3 DCA for solving problem (11)

DC Algorithm (DCA) has been introduced by Pham Dinh Tao in 1985 as an extension of the subgradient algorithm, and extensively developed by Le Thi Hoai An and Pham Dinh Tao since 1993 to solve DC programs. It is actually one of the rare algorithms for nonlinear nonconvex nonsmooth programming which allows solving very efficiently large-scale DC programs. DCA has successfully been applied in solving real world nonconvex programs to which it quite often gives global solutions and is proved to be more robust and more efficient than the related standard methods, especially for large-scale problems. For more details of DC programs and DCA, the reader is referred to [1–4] and the references therein.

According to the general framework of DCA, we need constructing two sequences $\{X^k\}$ and $\{Y^k\}$. In our problem, $\{X^k := (x^k, y^k)\}$ and $\{Y^k := (u^k, v^k)\}$. In order to compute $Y^k = (u^k, v^k)$, we need computing subdifferential of the function h at the point $X^k = (x^k, y^k)$, denoted by $\partial h(x^k, y^k)$.

Definition 2. (See [1, 2].) Let $\Gamma_0(\mathbb{R}^n)$ denote the convex cone of all lower semicontinuous proper convex functions on $\mathbb{R}^n$. For all $\theta \in \Gamma_0(\mathbb{R}^n)$ and $x_0 \in dom(\theta) := \{x \in \mathbb{R}^n : \theta(x) < +\infty\}$, $\partial\theta(x_0)$ denotes the subdifferential of θ at x_0

$$\partial\theta(x_0) := \{y \in \mathbb{R}^n : \theta(x) \geq \theta(x_0) + < x - x_0, y >, \forall x \in \mathbb{R}^n\}.$$

It is well-known that if θ is differentiable at x_0, then $\partial\theta(x_0)$ reduces to a singleton which is exactly $\{\nabla\theta(x_0)\}$.

In our problem, the convex function h was defined as $h(x,y) := (2\pi^2 t y^T y - tp(y) - d^T y) - c^T x$ which is a twice continuously differentiable function. Thus, $\partial h(x^k, y^k) = \{\nabla_{(x,y)} h(x^k, y^k)\}$. The vector $Y^k = (u^k, v^k)$ can be computed explicitly using the following equivalence:

$$(u^k, v^k) \in \partial h(x^k, y^k) \Leftrightarrow (u^k = -c, v^k = 4\pi^2 t y^k - 2\pi t \sin 2\pi y^k - d). \tag{12}$$

Let $g^*(y) := \sup\{< x, y > -g(x) : x \in \mathbb{R}^n\}$ be the conjugate function of g. For computing $X^{k+1} = (x^{k+1}, y^{k+1}) \in \partial g^*(u^k, v^k)$, we have to solve the convex quadratic program:

$$\min\{2\pi^2 t y^T y - \langle (x,y), (u^k, v^k)\rangle : (x,y) \in \mathbf{K}\}. \tag{13}$$

Every optimal solution of the problem (13) gives us one vector $X^{k+1} = (x^{k+1}, y^{k+1})$. Repeating the above operation, we can establish the sequences $\{X^k\}$ and $\{Y^k\}$.

DC Algorithm (DCA)

Initialization:
Choose an initial point $X^0 = (x^0, y^0) \in \mathbb{R}^n \times \mathbb{R}^m$.
Let t be a large enough positive number.
Let ϵ_1, ϵ_2 be sufficiently small positive numbers.
Iteration number $k = 0$.

Repeat:
- Calculate $(u^k, v^k) \in \partial h(x^k, y^k)$ via (12).
- Solve the quadratic convex program (13) to obtain (x^{k+1}, y^{k+1}).
- $k \leftarrow k + 1$.

Until:
If either $||X^k - X^{k-1}|| \le \epsilon_1(1 + ||X^{k-1}||)$
or $|F_t(X^k) - F_t(X^{k-1})| \le \epsilon_2(1 + |F_t(X^{k-1})|)$
Then **STOP** and verify:
If y^k is in a $\frac{1}{5}$ - cubic neighborhood of an integer point $\tilde{y} \in \mathbb{Z}_+^m$ and $(x^k, \tilde{y})$ is a feasible solution to the problem (1)
Then $(x^k, \tilde{y})$ is a feasible computed solution
Else (x^k, y^k) is the computed solution.

The convergence of DCA can be summarized in the next theorem whose proof is essentially based on the convergence theorem of the general scheme of DCA (see [1–3]).

Theorem 2 (Convergence properties of DC Algorithm).

1. *DCA generates a sequence $\{(x^k, y^k)\}$ such that the sequence $\{F_t(x^k, y^k)\}$ is decreasing and bounded below.*

2. If the optimal value of (11) is finite and the infinite sequences $\{X^k\}$ and $\{Y^k\}$ are bounded, then every limit point X^∞ of the sequence $\{X^k\}$ is a Karush-Kuhn-Tucker point.

4 A combination of DCA with a B&B scheme

In order to evaluate the quality of the solution obtained by DCA and improve the computed solution of DCA for finding a global optimal solution, we propose a hybrid method which combines DCA with an adapted Branch-and-Bound scheme for globally solving MILP.

Branching Suppose that we have already found a computed solution by DCA, denoted by (x^*, y^*). If it is not feasible solution of MILP, then we can find an element $y_i^* \notin \mathbb{Z}_+$, and the subdivision is performed in the way that $y_i \leq \lfloor y_i^* \rfloor$ or $y_i \geq \lceil y_i^* \rceil$(where $\lfloor y_i^* \rfloor$ (resp. $\lceil y_i^* \rceil$) means the floor (resp. ceil) number of y_i^*). The two subdivision problems can be described as

$$\min\{F_t(x,y) = g(y) - h(x,y) : (x,y) \in \mathbf{K}, y_i \leq \lfloor y_i^* \rfloor\}. \tag{14}$$

$$\min\{F_t(x,y) = g(y) - h(x,y) : (x,y) \in \mathbf{K}, y_i \geq \lceil y_i^* \rceil\}. \tag{15}$$

Note that (14) and (15) are also DC programs. The feasible sets of these problems are disjunctive, and the union of the two feasible sets might be smaller than $\mathbf{K}$, but it includes every feasible solution of the original MILP problem (1).

Bounding Solving directly the subproblems (14) and (15) is a difficult task and there is no extraordinary efficient and practical solution method. So we establish their lower bound problems:

$$\min\{f(x,y) = c^T x + d^T y : (x,y) \in \mathbf{K}, y_i \leq \lfloor y_i^* \rfloor\}. \tag{16}$$

$$\min\{f(x,y) = c^T x + d^T y : (x,y) \in \mathbf{K}, y_i \geq \lceil y_i^* \rceil\}. \tag{17}$$

Note that, the problem (16) (resp. (17)) is a lower bound problem to (14) (resp. (15)), because $F_t(x,y) = c^T x + d^T y + tp(y) \geq c^T x + d^T y$ for all $(x,y) \in \mathbf{K}$. It is clear that (16) and (17) are linear programs, so there are efficient practical algorithms for solving them even if they are large-scales. After solving a lower bound problem, the best current upper bound solution will be updated if a better feasible solution was discovered.

DCA + B&B for solving MILP The B&B method forms a search tree in a way that we always create branches with a node who has a smallest lower bound (a node here means a subdivision problem). During the search process, we can prune away every node who is an infeasible problem or whose lower bound is greater than the best current upper bound.

The algorithm terminates when every node of the search tree was pruned away or the gap between the best upper bound and the current minimal lower bound is less than a given tolerance.

When DCA is restarted?
A suggestion to restart DCA in some steps of B&B helps to reduce the computational time. We have several heuristic strategies to restart DCA:

- *The first strategy*: When a best current upper bound solution is discovered, we can restart DCA from this upper bound solution. Perhaps this strategy will find a better feasible solution. If a better feasible solution was found by DCA, the best current upper bound solution will be updated.

- *The second strategy*: When we choose a current minimal lower bound node to create branches, we can restart DCA from this best lower bound solution. Perhaps this strategy will find a better feasible solution for updating the best current upper bound.

Based on the above discussions, we have established a Global Optimization Algorithm, denoted by GOA-DCA, for solving general MILP. Because the length of this article is limited, for more details about GOA-DCA, the reader is referred to the technical report [9].

Note that, using this solution method for solving ILP, we just need to ignore the coefficients of the continuous variables, i.e. the vectors c, lb_x, ub_x and the matrices A, A_{eq} in the problem (1).

5 Computational Experiments

We have implemented the algorithm DCA and GOA-DCA in MATLAB R2007a and tested it on a laptop equipped with Windows XP Professional, Genuine Intel(R) CPU T2130 1.86GHz, 2G RAM. The version of C++ has also been implemented using CPLEX and COIN-OR.

In this section, we present an illustrative example of an integer linear program. More tests on MILP and ILP (medium-scales or large-scales) have been realized with a series of data from MIPLIB 3.0 [10], they are real world pure or mixed integer linear problems, some of them are large-scales ($10^5 - 10^6$ integer variables) and difficult to be globally solved.
An illustrative example to an integer linear program:
This integer linear program is defined as

$$\min\{d^T y : By \leq b, lb_y \leq y \leq ub_y, y \in \mathbb{Z}_+^m\}. \tag{18}$$

- Number of integer variables: $m = 12$
- Number of linear inequality constraints: 7

$d = -(96, 76, 56, 11, 86, 10, 66, 86, 83, 12, 9, 81)^T$

$$\mathrm{B} = \begin{bmatrix} 19 & 1 & 10 & 1 & 1 & 14 & 152 & 11 & 1 & 1 & 1 & 1 \\ 0 & 4 & 53 & 0 & 0 & 80 & 0 & 4 & 5 & 0 & 0 & 0 \\ 4 & 660 & 3 & 0 & 30 & 0 & 3 & 0 & 4 & 90 & 0 & 0 \\ 7 & 0 & 18 & 6 & 770 & 330 & 7 & 0 & 0 & 6 & 0 & 0 \\ 0 & 20 & 0 & 4 & 52 & 3 & 0 & 0 & 0 & 5 & 4 & 0 \\ 0 & 0 & 40 & 70 & 4 & 63 & 0 & 0 & 60 & 0 & 4 & 0 \\ 0 & 32 & 0 & 0 & 0 & 5 & 0 & 3 & 0 & 660 & 0 & 9 \end{bmatrix}$$

$b = (18209, 7692, 1333, 924, 26638, 61188, 13360)^T$

$lb_y = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)^T$

$ub_y = (26638, 26638, 26638, 26638, 26638, 26638, 26638, 26638, 26638, 26638, 26638, 26638)^T$

Using the Branch-and-Bound method for solving (18):

- The global optimal solution was found after 182 iterations.
- The global optimal value: -261922.
- The global optimal solution: [0,0,0,154,0,0,0,913,333,0,6499,1180].
- The average CPU time: 13.12 seconds.

Using DCA for solving (18):
We have tested the performance of DCA with different starting points. Here are some numerical results in Table 1:

Table 1. DCA test results for (18) with $\epsilon_1 = \epsilon_2 = 1e-3$ and $t = 1000$

TestNo	Starting point	#Iter	CPU time (secs.)	Obj (Min)	Integer solution (T/F)
1	$ub - lb$	30	0.5183	-261890	T
2	$\frac{ub-lb}{2}$	30	0.5079	-261890	T
3	$\frac{ub-lb}{3}$	30	0.4950	-261890	T
4	$\frac{ub-lb}{4}$	30	0.4913	-261890	T
5	$\frac{ub-lb}{5}$	30	0.5165	-261890	T
6	$\frac{ub-lb}{6}$	30	0.5306	-261890	T
7	$\frac{ub-lb}{8}$	30	0.5505	-261890	T
8	$\frac{ub-lb}{9}$	11	0.2583	-261657.341	F
9	$\frac{ub-lb}{20}$	21	0.3978	-249604.313	F
10	$\frac{ub-lb}{50}$	12	0.2396	-230200	T
11	$\frac{ub-lb}{100}$	10	0.2085	-137045	T

Table 1 shows:

- DCA often gives a feasible solution (TestNo 1-7, 10 and 11).
- Using different staring points, DCA often converges with a small number of iterations (less than 30 iterations).
- Most of the objective values in Table 1 are the number -261890. This number is close to the global optimal value which is -261922.

We calculate the relative error by the formula:

$$err := \frac{|\text{optimal value obtained by DCA} - \text{global optimal value}|}{|\text{global optimal value}|}.$$

We get $err = |-261890 + 261922|/261922 \approx 1.2E-4$. Such a small error shows the computed solutions of DCA are often close to the global optimal solution.

More tests for large-scale problems show that the good performance of DCA is not particular to this illustrative example, but a universal result. Especially, the iteration numbers are often relatively small (less than 60 for the tests with $10^5 - 10^6$ variables).

Using GOA-DCA for solving (18):

Here are some test results of GOA-DCA without restarting DCA during the B&B process:

- The optimal solution was discovered after 36 iterations.
- The optimal value: -261922.
- The optimal solution: [0,0,0,154,0,0,0,913,333,0,6499,1180] (globally optimized).
- The CPU time is 2.35 seconds $\ll$ 13.12 seconds (by B&B).

More test results show that GOA-DCA can always find the global optimal solution if it exists, and the CPU time is always smaller than B&B. The superiority of GOA-DCA with respect to B&B increases with the dimension. An interesting issue is how to restart DCA. More tests for large-scale problems show that the two strategies for restarting DCA play a quite important role in GOA-DCA method. DCA often finds rapidly a feasible solution and it improves considerably the best current upper bound during the B&B process and therefore accelerates the convergence of GOA-DCA. For more discussions of the tests on DCA and GOA-DCA, the reader is referred to the technical report [9].

6 Conclusion and future work

In this paper, we have presented a new continuous nonconvex optimization approach based on DC programming and DCA for solving the general MILP problem. Using a special penalized function, we get a DC program. With a suitable penalty parameter and a good starting point, DCA generates a sequence for which the objective values decrease and converge very fast to a computed solution (which is often a feasible solution of MILP and close to a global optimal solution). Despite its local character, DCA shows once again, its robustness, reliability, efficiency and globality. The hybrid method GOA-DCA which combines DCA with an adapted Branch-and-Bound aims at checking the globality of DCA and globally solving the MILP problem efficiently. Preliminary numerical tests

show that GOA-DCA usually gives the global optimal solution much faster than B&B method.

This paper could be considered as a first step of using the DC Programming approach for solving general MILP and ILP problems. Some extensions of this solution method are in development. For instance, we are trying to integrate GOA-DCA with some cuts (Gomory's cut, lift-and-projet cut, knapsack cover, etc.). We also want to extend this approach to more difficult nonlinear MIPs and pure Integer Programs:

1. Pure Integer Nonlinear Program (convex objective function, and DC objective function).
2. Mixed Integer Quadratic Program (convex objective function, and nonconvex objective function).
3. Mixed Integer Nonlinear Program with DC objective function, etc.

Mixed Integer Quadratic Programs with binary integer variables have already been treated in another work [5]. Results concerning these extensions will be reported subsequently.

References

1. Pham Dinh, T., Le Thi, H.A.: DC Programming. Theory, Algorithms, Applications: The State of the Art. LMI INSA - Rouen, France (2002)
2. Pham Dinh, T., Le Thi, H.A.: Convex analysis approach to D.C. programming: Theory, Algorithms and Applications. Acta Mathematica Vietnamica, vol. 22, 287-367 (1997)
3. Pham Dinh, T., Le Thi, H.A.: The DC programming and DCA Revisited with DC Models of Real World Nonconvex Optimization Problems. Annals of Operations Research, vol. 133, 23-46 (2005)
4. Pham Dinh, T., Le Thi, H.A.: DC optimization algorithms for solving the trust region subproblem. SIAM J.Optimization, vol. 8, 476-507 (1998)
5. Niu, Y.S.: Programmation DC et DCA pour la gestion du portefeuille de risque de chute du cours sous des contraintes de transaction. LMI INSA - Rouen, France (2006)
6. Ge, R.P., Huang, C.B.: A Continuous Approach to Nonlinear Integer Programming. Applied Mathematics and Computation, vol. 34, 39-60 (1989)
7. Nemhauser, G.L., Wolsey, L.A.: Integer and Combinatorial Optimization. Wiley-Interscience Publication (1999)
8. Luenberger, D.G.: Linear and Nonlinear Programming. Second edition, pp. 366-380. Springer (2003)
9. Pham Dinh, T., Niu, Y.S.: DC Programming for Mixed-Integer Program. Technical report, LMI INSA - Rouen (2008)
10. MIPLIB 3.0, `http://miplib.zib.de/miplib3/miplib.html`
11. COIN-OR, `http://www.coin-or.org/`

Troisième partie

Programmation avec fonctions polynomiales

Chapitre 6

Gestion de portefeuille avec moments d'ordre supérieur

Computational Optimization and Applications manuscript No.
(will be inserted by the editor)

An Efficient DC Programming Approach for Portfolio Decision with Higher Moments

Yi-Shuai Niu · Pham Dinh Tao

Received: date / Accepted: date

Abstract [1] Portfolio selection with higher moments is a NP-hard nonconvex polynomial optimization problem. In this paper, we propose an efficient local optimization approach based on DC (Difference of Convex functions) programming - called DCA (DC Algorithm) - that consists of solving the nonconvex program by a sequence of convex ones. DCA will construct, in each iteration, a suitable convex quadratic subproblem which can be easily solved by explicit method, due to the proposed special DC decomposition. Computational results show that DCA almost always converges to global optimal solutions while comparing with the global optimization methods (Gloptipoly, Branch-and-Bound) and it outperforms several standard local optimization algorithms.

Keywords DC Programming · DCA · Polynomial Optimization · Higher Moment Portfolio

1 Introduction

According to the Markowitz standard mean-variance portfolio model (MV model [1, 2]), the investors aim at maximizing the expected wealth of the portfolio (mean return rate: first moment) and minimizing its risk (variance of the portfolio: second moment). However, the MV framework is a special case based on the assumption that the asset returns are Gaussian distributed. Usually, this assumption does not hold exactly in

Yi-Shuai Niu
Laboratory of Mathematics
National Institute for Applied Sciences
BP 08, Avenue de l'Université, 76801, Rouen, France
E-mail: niuyishuai@hotmail.com

Pham Dinh Tao
Laboratory of Mathematics
National Institute for Applied Sciences
BP 08, Avenue de l'Université, 76801, Rouen, France
E-mail: pham@insa-rouen.fr

[1] **To appear in Computation Optimization and Applications. Revised version**

financial market, because many return distributions in the market exhibit fat tails and asymmetry that can not be described by their mean-variance alone. In many cases, the tails significantly affect portfolio performance [3]. Harvey and Siddique [4] show that skewness in stock return is relevant to portfolio selection. They found that in the presence of positive skewness, investors may be willing to accept a negative expected return. Several other studies show that skewness and kurtosis are both important factors in asset pricing ([5,6]). Therefore, for a more realistic portfolio selection model, skewness (third moment) and kurtosis (fourth moment) must be considered. The first attempt to extend the classical mean-variance optimization to higher moments was done by Jean in early 1970s [7]. Later, more general and rigorous model based on Mean-Variance-Skewness-Kurtosis (MVSK model) has been presented by several authors (Athayde and Flôres [8], Harvey et al. [9] etc).

The introduction of skewness and kurtosis makes the problem cumbersome and hard to be solved, since the skewness and kurtosis functions are highly nonlinear that can exhibit multiple maxima and minima. The existing numerical methods for nonlinear polynomial optimization with SDP relaxation can hardly be used when we tackle a problem with many variables (such as the software Gloptipoly can only tackle this problem with less than 18 variables), since many nonzero moment coefficients lead the polynomial computation and SDP relaxation to be more complicated and very slow in computation. Classical nonlinear optimization methods, such as SQP, Newton's method, trust region method, could not often give good computed solution, since the quality of computed solutions depends on the choice of an initial point. However, a good initial point is often difficult to find, and it depends on the specific structure of the optimization problem. Although we could guess an initial point by using some heuristic methods, the quality of such a point is hard to be theoretically guaranteed. DCA is a deterministic iterative method. It constructs a suitable upper bound convex quadratic programming subproblem in each iteration. Thanks to the special structure of our DC decomposition, these subproblems can be transformed into projection problems which could be explicitly computed. These distinctive features make DCA possible to handle efficiently large-scale problems. Computational experiments show that DCA converges to global solutions while comparing it with global optimization algorithms (Gloptipoly, Branch-and-Bound (B&B) and the combined DCA-B&B) and that it outperforms several standard local algorithms as SQP and Trust Region Method.

The paper is organized as follows. In the next section, we present a generic higher moment portfolio selection model within the mean-variance-skewness-kurtosis framework. In Section 3, we investigate key properties of the polynomial objective function, which will lead to an appropriate equivalent DC program. DC programming and DCA are outlined in Section 4. The DCA for solving MVSK problem is described in Section 5. Finally computational experiments and some conclusions and related future works are reported in the last two sections.

2 Problem Formulation

In this section, we present the generic higher moment portfolio selection model. Consider a portfolio with n assets. The following notations will be used in this article:

Table 1 NOTATION

n	Total number of assets.
T	Total number of periods.
$\mathbf{x}$	The decision variables of a portfolio, a vector in $\mathbb{R}^n$.
x_i	The i-th element of $\mathbf{x}$, the percentage of wealth invested in the i-th risky asset.
R_{it}	The return rate on asset i in period t, where $i = 1, ..., n$ and $t = 1, ..., T$.
R_i	The return rate on the i-th asset, $i = 1, \ldots, n$.
$\mathbf{R}$	The return rate vector, i.e. $\mathbf{R} = (R_i)$.
μ_i	The mean return rate of asset i, i.e. $\mu_i = E(R_i)$.
$\boldsymbol{\mu}$	The vector of mean returns (μ_i), $i = 1, \ldots, n$.
σ_{ij}	The covariance between R_i and R_j.
V	The covariance matrix (σ_{ij}), $i, j = 1, \ldots, n$.
s_{ijk}	The co-skewness among R_i, R_j and R_k, $i, j, k = 1, \ldots, n$.
k_{ijkl}	The co-kurtosis among R_i, R_j, R_k and R_l, $i, j, k, l = 1, \ldots, n$.

Some parameters in table 1 could be computed as follows:

$$\mu_i := E[R_i] = \frac{1}{T}\sum_{t=1}^{T} R_{it}.$$

$$\sigma_{ij} := E[(R_i - \mu_i)(R_j - \mu_j)] = \frac{1}{T}\sum_{t=1}^{T}(R_{it} - \mu_i)(R_{jt} - \mu_j).$$

$$s_{ijk} := E[(R_i - \mu_i)(R_j - \mu_j)(R_k - \mu_k)] = \frac{1}{T}\sum_{t=1}^{T}(R_{it} - \mu_i)(R_{jt} - \mu_j)(R_{kt} - \mu_k).$$

$$\begin{aligned} k_{ijkl} &:= E[(R_i - \mu_i)(R_j - \mu_j)(R_k - \mu_k)(R_l - \mu_l)] \\ &= \frac{1}{T}\sum_{t=1}^{T}(R_{it} - \mu_i)(R_{jt} - \mu_j)(R_{kt} - \mu_k)(R_{lt} - \mu_l). \end{aligned}$$

In the higher moment portfolio selection model, the investor aims at maximizing the expected return and the skewness of the portfolio, and minimizing the variance and the kurtosis [33]. The reason is that the positive skewness is desirable (since it corresponds to higher returns albeit with low probability) while kurtosis is undesirable (since it implies that the investor is exposed to more risk) [32]. This multi-objective optimization problem can be reformulated as the following single-objective nonconvex optimization problem

$$\begin{array}{ll} \min & f(\mathbf{x}) = -\alpha E(\mathbf{x}) + \beta V(\mathbf{x}) - \gamma S(\mathbf{x}) + \delta K(\mathbf{x}) \\ s.t. & \sum_{i=1}^{n} x_i = 1, x_i \geq 0, i = 1, \ldots, n \end{array} \tag{1}$$

where $\alpha, \beta, \gamma, \delta$ denote the investor's preference levels corresponding to the four moments, $0 \leq \alpha, \beta, \gamma, \delta \leq 1$ and $\alpha + \beta + \gamma + \delta = 1$. This model is called *generic Mean-Variance-Skewness-Kurtosis based portfolio decision model*. In this model, we suppose no short-selling, so $x_i \geq 0, i = 1, \ldots, n$. The investor's preference parameters could be used to characterize the preference of the investor. For instance, $\alpha = 1$ means that the investor is risk-seeking, $\beta = 1$ implies the investor is risk-aversing.

The four moments of the portfolio could be computed by the following formulas:

$$Mean : E(\mathbf{x}) = \sum_{i=1}^{n} x_i \mu_i. \tag{2}$$

$$Variance : V(\mathbf{x}) = \sum_{i,j=1}^{n} \sigma_{ij} x_i x_j. \tag{3}$$

$$Skewness : S(\mathbf{x}) = \sum_{i,j,k=1}^{n} s_{ijk} x_i x_j x_k. \tag{4}$$

$$Kurtosis : K(\mathbf{x}) = \sum_{i,j,k,l=1}^{n} k_{ijkl} x_i x_j x_k x_l. \tag{5}$$

Specifically, the objective function f in (1) is a nonconvex, fourth-order continuous and differentiable, multivariate polynomial function. More properties about the objective function will be presented in the next section. The constraints is a standard $(n-1)-$simplex which is a nonempty polyhedral compact convex set of $\mathbb{R}^n$. So the difficulty of this problem lies in the nonconvex objective function.

3 DC Reformulation for MVSK model

3.1 Properties of the objective function

In this subsection, we present some properties on the gradient and the Hessian of the objective function f in (1) defined as

$$\begin{aligned} f(\mathbf{x}) &= -\alpha E(\mathbf{x}) + \beta V(\mathbf{x}) - \gamma S(\mathbf{x}) + \delta K(\mathbf{x}) \\ &= -\alpha \sum_{i=1}^{n} x_i \mu_i + \beta \sum_{i,j=1}^{n} \sigma_{ij} x_i x_j - \gamma \sum_{i,j,k=1}^{n} s_{ijk} x_i x_j x_k + \delta \sum_{i,j,k,l=1}^{n} k_{ijkl} x_i x_j x_k x_l. \end{aligned}$$

Thanks to the homogeneity of the components E, V, S and K, we can simplify the calculation of the gradient and the Hessian of f. The reason to calculate the gradient and the Hessian of f is due to need of DCA.

A function $g : \mathbb{R}^n \to \mathbb{R}$ is said to be homogeneous of degree k if $g(\alpha \mathbf{x}) = \alpha^k g(\mathbf{x})$ for all nonzero $\alpha \in \mathbb{R}$ and $\mathbf{x} \in \mathbb{R}^n$. It is easy to verify that the function E (resp. V, S and K) is homogeneous of degree 1 (resp. 2, 3 and 4).

Remark 1 Note that the objective function f is the sum of homogeneous functions. However, this function is no longer homogeneous unless 3 of the investor's preferences α,β,γ and δ are equal to zero.

The well-known Euler's theorem says that if $\phi : \mathbb{R}^n \to \mathbb{R}$ is differentiable and homogeneous of degree k, then $\langle \mathbf{x}, \nabla \phi(\mathbf{x}) \rangle = k\phi(\mathbf{x})$. We get inspiration from the Euler's theorem to establish the relation between $\nabla \phi(\mathbf{x})$ and $\nabla^2 \phi(\mathbf{x})$ as follows:

Lemma 1 *Suppose that the function $\phi : \mathbb{R}^n \to \mathbb{R}$ is a twice differentiable homogeneous function of degree k ($k \geq 1$), then*

$$(k-1)\nabla\phi(\mathbf{x}) = \nabla^2\phi(\mathbf{x})\mathbf{x}.$$

Proof Euler's theorem gives the equation

$$\langle \mathbf{x}, \nabla\phi(\mathbf{x})\rangle = k\phi(\mathbf{x}).$$

We differentiate this equation with respect to x_i for all $i = 1, \ldots, n$.

$$\frac{\partial}{\partial x_i}\langle \mathbf{x}, \nabla\phi(\mathbf{x})\rangle = k\frac{\partial}{\partial x_i}\phi(\mathbf{x}).$$

Let $\mathbf{u} := \mathbf{x}$, $\mathbf{v} := \nabla\phi(\mathbf{x})$ and $\theta(\mathbf{u}, \mathbf{v}) := \langle \mathbf{u}, \mathbf{v}\rangle$ we find by the chain rule that

$$\frac{\partial}{\partial x_i}\langle \mathbf{x}, \nabla\phi(\mathbf{x})\rangle = \frac{\partial}{\partial x_i}\theta(\mathbf{u}, \mathbf{v}) = \langle \frac{\partial\theta}{\partial\mathbf{u}}, \frac{\partial\mathbf{u}}{\partial x_i}\rangle + \langle \frac{\partial\theta}{\partial\mathbf{v}}, \frac{\partial\mathbf{v}}{\partial x_i}\rangle$$

where

$$\frac{\partial\theta}{\partial\mathbf{u}} = \mathbf{v}; \frac{\partial\theta}{\partial\mathbf{v}} = \mathbf{u}; \frac{\partial\mathbf{u}}{\partial x_i} = \mathbf{e_i}; \frac{\partial\mathbf{v}}{\partial x_i} = (\frac{\partial\phi}{\partial x_1\partial x_i}, \ldots, \frac{\partial\phi}{\partial x_n\partial x_i})^T$$

($\{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$ being the canonical basis of $\mathbb{R}^n$).

Hence

$$\frac{\partial}{\partial x_i}\theta(\mathbf{u}, \mathbf{v}) = \langle \mathbf{v}, \mathbf{e}_i\rangle + \langle \mathbf{u}, \frac{\partial\mathbf{v}}{\partial x_i}\rangle, i = 1, \ldots, n.$$

Therefore, we can compute $\nabla\langle \mathbf{x}, \nabla\phi(\mathbf{x})\rangle$ by

$$\begin{aligned}\nabla\langle \mathbf{x}, \nabla\phi(\mathbf{x})\rangle = \nabla\theta(\mathbf{u}, \mathbf{v}) &= (\langle \mathbf{v}, \mathbf{e_1}\rangle, \ldots, \langle \mathbf{v}, \mathbf{e_n}\rangle)^T + (\langle \mathbf{u}, \frac{\partial\mathbf{v}}{\partial x_1}\rangle, \ldots, \langle \mathbf{u}, \frac{\partial\mathbf{v}}{\partial x_n}\rangle)^T \\ &= \mathbf{v} + \nabla^2\phi(\mathbf{x})\mathbf{u} \\ &= \nabla\phi(\mathbf{x}) + \nabla^2\phi(\mathbf{x})\mathbf{x}.\end{aligned}$$

Finally, we have $\nabla\phi(\mathbf{x}) + \nabla^2\phi(\mathbf{x})\mathbf{x} = \nabla\langle \mathbf{x}, \nabla\phi(\mathbf{x})\rangle = k\nabla\phi(\mathbf{x})$.
The equation above can be written as the equation

$$\nabla^2\phi(\mathbf{x})\mathbf{x} = (k-1)\nabla\phi(\mathbf{x}).$$

□

The gradient and the Hessian of the objective function f can be computed via the following formulas:

$$\nabla f(\mathbf{x}) = -\alpha\nabla E(\mathbf{x}) + \beta\nabla V(\mathbf{x}) - \gamma\nabla S(\mathbf{x}) + \delta\nabla K(\mathbf{x})$$

$$\nabla^2 f(\mathbf{x}) = -\alpha\nabla^2 E(\mathbf{x}) + \beta\nabla^2 V(\mathbf{x}) - \gamma\nabla^2 S(\mathbf{x}) + \delta\nabla^2 K(\mathbf{x})$$

where $\nabla E(\mathbf{x}) = \boldsymbol{\mu}$ and $\nabla^2 E(\mathbf{x}) = \mathbf{0}_{n\times n}$. $\nabla V(\mathbf{x}) = 2\mathsf{V}\mathbf{x}$ with V being the covariance matrix (σ_{ij}), and $\nabla^2 V(\mathbf{x}) = 2\mathsf{V}$.
By virtue of Lemma 1, we have

$$\nabla S(\mathbf{x}) = \frac{\nabla^2 S(\mathbf{x})\mathbf{x}}{(3-1)} \text{ and } \nabla K(\mathbf{x}) = \frac{\nabla^2 K(\mathbf{x})\mathbf{x}}{(4-1)}.$$

The expression of $\nabla^2 S(\mathbf{x})$ and $\nabla^2 K(\mathbf{x})$ could be computed as follows:

Proposition 1 *$S(\mathbf{x}) = \sum_{i,j,k=1}^{n} s_{ijk}x_ix_jx_k$ being homogeneous of degree 3, and $K(\mathbf{x}) = \sum_{i,j,k,l=1}^{n} k_{ijkl}x_ix_jx_kx_l$ being homogeneous of degree 4. We can derive their Hessian matrices as*

1. $\nabla^2 S(\mathbf{x}) = (\frac{\partial^2 S(\mathbf{x})}{\partial x_i \partial x_j})$ *with* $\frac{\partial^2 S(\mathbf{x})}{\partial x_i \partial x_j} = 6\sum_{k=1}^{n} s_{ijk}x_k, \forall i, j = 1, \ldots, n$
2. $\nabla^2 K(\mathbf{x}) = (\frac{\partial^2 K(\mathbf{x})}{\partial x_i \partial x_j})$ *with* $\frac{\partial^2 K(\mathbf{x})}{\partial x_i \partial x_j} = 12\sum_{k,l=1}^{n} k_{ijkl}x_kx_l, \forall i, j = 1, \ldots, n.$

Proof Firstly, for any fixed index $i = 1, \ldots, n$, we should sort the expression $S(\mathbf{x}) := \sum_{i,j,k=1}^{n} s_{ijk}x_ix_jx_k$ with respect to x_i in a descending order as

$$S(\mathbf{x}) = s_{iii}x_i^3 + 3x_i^2 \sum_{k\neq i} s_{iik}x_k + 3x_i \sum_{k,j\neq i} s_{ijk}x_jx_k + r(\mathbf{x})$$

where $r(\mathbf{x})$ does not consist of x_i.
Then, we can derive the expression of $\frac{\partial S}{\partial x_i}$, $\frac{\partial^2 S}{\partial x_i^2}$ and $\frac{\partial^2 S}{\partial x_i \partial x_j}$ as follows:

$$\frac{\partial S(\mathbf{x})}{\partial x_i} = 3s_{iii}x_i^2 + 6x_i \sum_{k\neq i} s_{iik}x_k + 3 \sum_{k,j\neq i} s_{ijk}x_jx_k \tag{6}$$

so that,

$$\frac{\partial^2 S(\mathbf{x})}{\partial x_i^2} = 6s_{iii}x_i + 6\sum_{k\neq i} s_{iik}x_k = 6\sum_{k=1}^{n} s_{iik}x_k.$$

In order to derive the expression of $\frac{\partial^2 S}{\partial x_i \partial x_j}$ with $j \neq i$, we should sort $\frac{\partial S}{\partial x_i}$ with respect to x_j in a descending order as follows

$$\frac{\partial S(\mathbf{x})}{\partial x_i} = 3s_{ijj}x_j^2 + 6x_j \sum_{i,k\neq j} s_{ijk}x_k + \bar{r}(\mathbf{x})$$

where $\bar{r}(\mathbf{x})$ does not consist of x_j.
Hence,

$$\begin{aligned}\frac{\partial^2 S(\mathbf{x})}{\partial x_i \partial x_j} &= 6s_{ijj}x_j + 6\sum_{i,k\neq j} s_{ijk}x_k \\ &= 6\sum_{i\neq j} s_{ijk}x_k = 6\sum_{k=1}^{n} s_{ijk}x_k.\end{aligned}$$

Thus, we have $\frac{\partial^2 S(\mathbf{x})}{\partial x_i \partial x_j} = 6\sum_{k=1}^{n} s_{ijk}x_k$ for all $i, j = 1, \ldots, n$.
Finally, we proved $\nabla^2 S(\mathbf{x}) := (\frac{\partial^2 S(\mathbf{x})}{\partial x_i \partial x_j})$ with $\frac{\partial^2 S(\mathbf{x})}{\partial x_i \partial x_j} = 6\sum_{k=1}^{n} s_{ijk}x_k$.
The expression of $\nabla^2 K(\mathbf{x})$ could be found in a similar way.

□

In fact, one can use the expressions of the gradient and the Hessian in Proposition 1 to verify Lemma 1. The Proposition 1 provides an explicit formula for computing the Hessian of the functions S and K; while Lemma 1 gives the relation between the gradient and the Hessian.

We must emphasize that, the calculation of gradients (∇S and ∇K) needs more operations than that of Hessian ($\nabla^2 S$ and $\nabla^2 K$). Normally, this feature seems incorrect, since the gradient needs computing n elements while the Hessian needs computing n^2 elements. In order to prove this special feature, let us compute their operations.

The explicit formula of ∇S was given by (6) as

$$\nabla S(\mathbf{x}) = (\frac{\partial S(\mathbf{x})}{\partial x_i}) = (3s_{iii}x_i^2 + 6x_i \sum_{k\neq i} s_{iik}x_k + 3 \sum_{k,j\neq i} s_{ijk}x_j x_k), i = 1, \ldots, n$$

while the formula of $\nabla^2 S$ in Proposition 1 is

$$\nabla^2 S(\mathbf{x}) = (\frac{\partial^2 S(\mathbf{x})}{\partial x_i \partial x_j}) \text{ with } \frac{\partial^2 S(\mathbf{x})}{\partial x_i \partial x_j} = 6 \sum_{k=1}^{n} s_{ijk}x_k, \forall i, j = 1, \ldots, n.$$

For computing each element of $\frac{\partial S}{\partial x_i}$ requires $2n^2 - 3n + 7$ multiplications and $n^2 - n - 2$ subtractions, for total of $3n^2 - 4n + 5$ operations. Therefore, n elements of gradient need $3n^3 - 4n^2 + 5n$ operations. However, for computing each Hessian's element $\frac{\partial^2 S}{\partial x_i \partial x_j}$ requires $n + 1$ multiplications and $n - 1$ subtractions, for total of $2n$ operations. So n^2 elements of Hessian need $2n^3$ operations.

The inequality $3n^3 - 4n^2 + 5n > 2n^3$ holds for every $n \in \mathbb{N}$. Therefore, computing a gradient vector ∇S needs more operations than computing a Hessian matrix $\nabla^2 S$. We have a similar result for the function K.

This particular feature is due to the fact that the expression of gradient is more complicated than that of Hessian. On the other hand, if we use Lemma 1 for computing the gradient from the expression of Hessian, we will have less operation than using the explicit formula (6). For instance, we can derive ∇S from $\nabla^2 S$ via the equation $\nabla S(\mathbf{x}) = \nabla^2 S(\mathbf{x})\mathbf{x}/2$, then we have total $2n^3 + 2n^2$ operations in computing ∇S. Comparing with the formula (6), we found that, when $n > 5$, using Lemma 1 to calculate ∇S cost less operation than using the explicit formula (6), since the inequality $3n^3 - 4n^2 + 5n > 2n^3 + 2n^2$ holds for $n > 5$. Moreover, from the computational viewpoint, the equation in Lemma 1 needs only matrix multiplications. Many softwares and packages for matrix operations, such as MATLAB, SCILAB, LAPACK, BLAS etc, can be used conveniently and compute efficiently.

Finally, we have the expressions of gradient and Hessian of f as

$$\begin{aligned} \nabla f(\mathbf{x}) &= -\alpha \nabla E(\mathbf{x}) + \beta \nabla V(\mathbf{x}) - \gamma \nabla S(\mathbf{x}) + \delta \nabla K(\mathbf{x}) \\ &= -\alpha \boldsymbol{\mu} + 2\beta \mathsf{V}\mathbf{x} - \frac{\gamma \nabla^2 S(\mathbf{x})\mathbf{x}}{2} + \frac{\delta \nabla^2 K(\mathbf{x})\mathbf{x}}{3} \end{aligned} \tag{7}$$

$$\begin{aligned} \nabla^2 f(\mathbf{x}) &= -\alpha \nabla^2 E(\mathbf{x}) + \beta \nabla^2 V(\mathbf{x}) - \gamma \nabla^2 S(\mathbf{x}) + \delta \nabla^2 K(\mathbf{x}) \\ &= 2\beta \mathsf{V} - \gamma \nabla^2 S(\mathbf{x}) + \delta \nabla^2 K(\mathbf{x}) \end{aligned} \tag{8}$$

where $\nabla^2 S(\mathbf{x}) = (\frac{\partial^2 S(\mathbf{x})}{\partial x_i \partial x_j})$ with $\frac{\partial^2 S(\mathbf{x})}{\partial x_i \partial x_j} = 6 \sum_{k=1}^{n} s_{ijk}x_k$ and $\nabla^2 K(\mathbf{x}) = (\frac{\partial^2 K(\mathbf{x})}{\partial x_i \partial x_j})$ with $\frac{\partial^2 K(\mathbf{x})}{\partial x_i \partial x_j} = 12 \sum_{k,l=1}^{n} k_{ijkl}x_k x_l$, for all $i, j = 1, \ldots, n$.

3.2 DC Reformulation

Let $S_{n-1} := \{\sum_{i=1}^{n} x_i = 1, x_i \geq 0, i = 1, \ldots, n\}$ be the standard $(n-1)$-simplex. In order to establish a suitable DC (difference of two convex functions) decomposition to f, we can represent f as

$$f(\mathbf{x}) = (\frac{\eta}{2}\mathbf{x}^T\mathbf{x}) - (\frac{\eta}{2}\mathbf{x}^T\mathbf{x} - f(\mathbf{x})).$$

When the parameter η takes a suitable value, the expression above becomes a DC function. In fact, we can take $\eta \geq \rho(\nabla^2 f(\mathbf{x}))$ for all $x \in S_{n-1}$, where $\rho(\nabla^2 f(\mathbf{x}))$ denotes the spectral radius of $\nabla^2 f(\mathbf{x})$, and then the function $(\frac{\eta}{2}\mathbf{x}^T\mathbf{x}) - (\frac{\eta}{2}\mathbf{x}^T\mathbf{x} - f(\mathbf{x}))$ should be a locally DC decomposition on S_{n-1}. The reason is that $\frac{\eta}{2}\mathbf{x}^T\mathbf{x}$ and $\frac{\eta}{2}\mathbf{x}^T\mathbf{x} - f(\mathbf{x})$ are convex functions on S_{n-1}, since their Hessian matrices are semi-positive definite when $\eta \geq \rho(\nabla^2 f(\mathbf{x}))$, $\forall \mathbf{x} \in S_{n-1}$.

The analysis above shows that, in order to find a suitable value for η, we can solve the optimization problem $\eta^* := max\{\rho(\nabla^2 f(\mathbf{x})) : \mathbf{x} \in S_{n-1}\}$, and then take $\eta \geq \eta^*$. However, in this problem, we only need to find an upper estimation of $\rho(\nabla^2 f(\mathbf{x}))$ on S_{n-1}, since we can easily compute an upper estimation via the following inequality $\rho(\nabla^2 f(\mathbf{x})) \leq \| \nabla^2 f(\mathbf{x}) \|_\infty$, where $\| A \|_\infty$ denotes the infinity norm of the matrix A defined as $\| A \|_\infty := \max_{1\leq i\leq n}(\sum_{j=1}^{n} |a_{ij}|)$, the maximum absolute row sum of A. Thus, we can prove the following proposition:

Proposition 2

$$\rho(\nabla^2 f(\mathbf{x})) \leq 2 \| \beta \mathsf{V} \|_\infty + 6|\gamma| \max_{1\leq i\leq n}(\sum_{j,k=1}^{n} |s_{ijk}|) + 12|\delta| \max_{1\leq i\leq n}(\sum_{j,k,l=1}^{n} |k_{ijkl}|) \quad (9)$$

for all $x \in S_{n-1}$.

Proof As we know,

$$\begin{aligned}\rho(\nabla^2 f(\mathbf{x})) &\leq \| \nabla^2 f(\mathbf{x}) \|_\infty = \| 2\beta\mathsf{V} - \gamma\nabla^2 S(\mathbf{x}) + \delta\nabla^2 K(\mathbf{x}) \|_\infty \\ &\leq 2|\beta| \| \mathsf{V} \|_\infty + |\gamma| \| \nabla^2 S(\mathbf{x}) \|_\infty + |\delta| \| \nabla^2 K(\mathbf{x}) \|_\infty .\end{aligned}$$

Then we need to compute respectively $\| \nabla^2 S(\mathbf{x}) \|_\infty$ and $\| \nabla^2 K(\mathbf{x}) \|_\infty$.
With respect to $\| \nabla^2 S(\mathbf{x}) \|_\infty$, we have

$$\begin{aligned}\| \nabla^2 S(\mathbf{x}) \|_\infty &= \max_{1\leq i\leq n} \sum_{j=1}^{n} |6 \sum_{k=1}^{n} s_{ijk} x_k| \\ &\leq 6 \max_{1\leq i\leq n} \sum_{j=1}^{n} \sum_{k=1}^{n} |s_{ijk}||x_k|.\end{aligned}$$

Since $\forall x \in S_{n-1} \subset [0,1]^n$, we have $|x_i| \leq 1, i = 1, \ldots, n$. Therefore,

$$\| \nabla^2 S(\mathbf{x}) \|_\infty \leq 6 \max_{1\leq i\leq n} \sum_{j=1}^{n} \sum_{k=1}^{n} |s_{ijk}|.$$

With respect to $\| \nabla^2 K(\mathbf{x}) \|_\infty$, we have

$$\begin{aligned} \| \nabla^2 K(\mathbf{x}) \|_\infty &= \max_{1\leq i\leq n} \sum_{j=1}^{n} |12 \sum_{k,l=1}^{n} k_{ijkl} x_k x_l| \\ &\leq 12 \max_{1\leq i\leq n} \sum_{j=1}^{n} \sum_{k,l=1}^{n} |k_{ijkl}||x_k||x_l| \\ &\leq 12 \max_{1\leq i\leq n} \sum_{j=1}^{n} \sum_{k,l=1}^{n} |k_{ijkl}|. \end{aligned}$$

Finally, we have

$$\begin{aligned} \rho(\nabla^2 f(\mathbf{x})) &\leq 2|\beta| \| \mathsf{V} \|_\infty + |\gamma| \| \nabla^2 S(\mathbf{x}) \|_\infty + |\delta| \| \nabla^2 K(\mathbf{x}) \|_\infty \\ &\leq 2|\beta| \| \mathsf{V} \|_\infty + 6|\gamma| \max_{1\leq i\leq n} (\sum_{j,k=1}^{n} |s_{ijk}|) + 12|\delta| \max_{1\leq i\leq n} (\sum_{j,k,l=1}^{n} |k_{ijkl}|). \end{aligned}$$

□

Let us choose the value of η as

$$\eta = 2|\beta| \| \mathsf{V} \|_\infty + 6|\gamma| \max_{1\leq i\leq n} (\sum_{j,k=1}^{n} |s_{ijk}|) + 12|\delta| \max_{1\leq i\leq n} (\sum_{j,k,l=1}^{n} |k_{ijkl}|). \tag{10}$$

Therefore, we can reformulate the initial optimization problem (1) as a DC program

$$\begin{array}{ll} \min & f(\mathbf{x}) = g(\mathbf{x}) - h(\mathbf{x}) \\ s.t. & \mathbf{x} \in S_{n-1} \end{array} \tag{11}$$

where the function $g(\mathbf{x}) := \frac{\eta}{2}\mathbf{x}^T\mathbf{x}$ is convex on $\mathbb{R}^n$, and $h(\mathbf{x}) := \frac{\eta}{2}\mathbf{x}^T\mathbf{x} - f(\mathbf{x})$ is convex on S_{n-1}.

4 DC Programming and DCA

DC programming and DCA (DC Algorithms) were first introduced by Pham Dinh Tao in their preliminary form in 1985. They have been extensively developed since 1994 by Le Thi Hoai An and Pham Dinh Tao, see e.g. [10]-[25] and also the webpage `http://lita.sciences.univ-metz.fr/~lethi/`. DCA have been successfully applied to many large-scale (smooth or nonsmooth) nonconvex programs in different fields of applied sciences for which they often give global solutions. The algorithms have proved to be more robust and efficient than the standard methods.

To give the reader an easy understanding of the theory of DC programming and DCA and of our motivation to introduce interior point techniques to DCA for solving large-scale nonconvex quadratic programs, we first briefly outline these tools and then apply them to nonconvex quadratic programs reformulated as DC programs. DCA's general convergence properties will be specialized for this class of DC programs.

4.1 General DC programming

Let $X = \mathbb{R}^n$ and $\|.\|$ be the Euclidean norm on X and let $\Gamma_0(\mathrm{X})$ denote the convex cone of all lower semi-continuous and proper (i.e. not identically equal to $+\infty$) convex functions defined on X and taking values in $\mathbb{R} \cup \{+\infty\}$. The dual space of X denotes by Y which could be identified with X itself. For $\theta \in \Gamma_0(\mathrm{X})$, the effective domain of θ, denoted dom θ, is defined by

$$\text{dom } \theta := \{x \in \mathrm{X} : \theta(x) < +\infty\}.$$

A general DC program is of the form

$$(P_{dc}) \quad \alpha = \inf\{f(x) := g(x) - h(x) \ : \ x \in \mathrm{X}\},$$

with $g, h \in \Gamma_0(\mathrm{X})$. Such a function f is called a DC function, and $g - h$, a DC decomposition of f while the convex functions g and h are DC components of f. In DC programming [10,11,14], the convention

$$(+\infty) - (+\infty) := +\infty \tag{12}$$

has been adopted to avoid the ambiguity on the determination of $(+\infty) - (+\infty)$. Such a case does not present any interest and can be discarded. In fact, we are actually concerned with the following problem

$$\alpha = \inf\{f(x) := g(x) - h(x) \ : \ x \in \text{dom } h\},$$

which is equivalent to (P_{dc}) under the convention (12). Remark that if the optimal value α is finite then dom $g \subset$ dom h. It should be noted that a constrained DC program (C being a nonempty closed convex set)

$$\alpha = \inf\{\varphi(x) - \psi(x) \ : \ x \in C\}$$

is equivalent to the unconstrained DC program by adding the indicator function χ_C of C ($\chi_C(x) = 0$ if $x \in C$, $+\infty$ otherwise) to the first DC component φ :

$$\alpha = \inf\{g(x) - h(x) \ : \ x \in \mathrm{X}\},$$

where $g := \varphi + \chi_C$ and $h := \psi$.Let

$$g^*(y) := \sup\{\langle x, y\rangle - g(x) \ : \ x \in \mathrm{X}\}$$

be the conjugate function of g. By using the fact that every function $h \in \Gamma_0(\mathrm{X})$ is characterized as a pointwise supremum of a collection of affine functions, say

$$h(x) := \sup\{\langle x, y\rangle - h^*(y) : y \in \mathrm{Y}\},$$

we have

$$\alpha = \inf\{h^*(y) - g^*(y) : y \in \text{dom } h^*\},$$

that is written, in virtue of the convention (12):

$$(D_{dc}) \quad \alpha = \inf\{h^*(y) - g^*(y) : y \in \mathrm{Y}\},$$

which is the dual DC program of (P_{dc}).Finally the finiteness of α implies both inclusions dom $g \subset$ dom h and dom $h^* \subset$ dom g^*. We will denote by $\mathcal{P}$ and $\mathcal{D}$ be the

solution sets of (P_{dc}) and (D_{dc}) respectively.

Recall that, for $\theta \in \Gamma_0(\mathrm{X})$ and $x_0 \in \mathrm{dom}\ \theta$, $\partial\theta(x_0)$ denotes the subdifferential of θ at x_0, *i.e.,*

$$\partial\theta(x_0) := \{y \in \mathrm{Y} : \theta(x) \geq \theta(x_0) + \langle x - x_0, y\rangle,\ \forall x \in \mathrm{X}\} \tag{13}$$

(see [26,27]). The subdifferential $\partial\theta(x_0)$ is a closed convex set in Y. It generalizes the derivative in the sense that θ is differentiable at x_0 if and only if $\partial\theta(x_0)$ is a singleton, which is exactly $\{\nabla\theta(x_0)\}$.

The domain of $\partial\theta$, denoted dom $\partial\theta$, is defined by: dom $\partial\theta := \{x \in \mathrm{dom}\ \theta : \partial\theta(x) \neq \emptyset\}$.

$$ri(\mathrm{dom}\ \theta) \subset \mathrm{dom}\ \partial\theta \subset \mathrm{dom}\ \theta \tag{14}$$

where ri stands for the relative interior.

A function $\phi \in \Gamma_0(X)$ is a polyhedral convex function if it can be expressed as

$$\phi(x) = \sup\{\langle a_i, x\rangle - \gamma_i : i = 1, \ldots, m\} + \chi_K(x)$$

where $a_i \in Y$, $\gamma_i \in \mathbb{R}$ for $i = 1, \ldots, m$ and K is a nonempty polyhedral convex set in X (see [26]). A DC program is called polyhedral if either g or h is a polyhedral convex function. This class of DC programs, which is frequently encountered in real-life optimization problems and was extensively developed in our previous works (see e.g. [17] and references therein), enjoys interesting properties (from both theoretical and practical viewpoints) concerning the local optimality and the finite convergence of DCA.

DC programming investigates the structure of the vector space $DC(\mathrm{X}) := \Gamma_0(\mathrm{X}) - \Gamma_0(\mathrm{X})$, DC duality and optimality conditions for DC programs. The complexity of DC programs resides, of course, in the lack of verifiable conditions for global optimality.

We developed instead the following necessary local optimality conditions for DC programs in their primal part, by symmetry their dual part is trivial (see ([10,11,14] and references therein):

$$\partial h(x^*) \cap \partial g(x^*) \neq \emptyset \tag{15}$$

(such a point x^* is called critical point of $g - h$ or generalized KKT point for (P_{dc})), and

$$\emptyset \neq \partial h(x^*) \subset \partial g(x^*). \tag{16}$$

The condition (16) is also sufficient (for local optimality) in many important classes of DC programs. In particular it is sufficient for the next cases quite often encountered in practice ([10,11,14] and references therein):

- In polyhedral DC programs with h being a polyhedral convex function . In this case, if h is differentiable at a critical point x^*, then x^* is actually a local minimizer for (P_{dc}). Since a convex function is differentiable everywhere except for a set of measure zero, one can say that a critical point x^* is almost always a local minimizer for (P_{dc}).
- In case the function f is locally convex at x^*. Note that, if h is polyhedral convex, then $f = g - h$ is locally convex everywhere h is differentiable.

The transportation of global solutions between (P_{dc}) and (D_{dc}) can be expressed by:

$$\bigcup_{y^* \in \mathcal{D}} \partial g^*(y^*) \subset \mathcal{P}, \ \bigcup_{x^* \in \mathcal{P}} \partial h(x^*) \subset \mathcal{D}. \tag{17}$$

Moreover, equality holds in the first inclusion of (17) if $\mathcal{P} \subseteq \text{dom } \partial h$, in particular if $\mathcal{P} \subseteq \text{ri}(\text{dom } h)$ according to (14). Similar property relative to the second inclusion can be stated by duality. On the other hand, under technical conditions this transportation holds also for local solutions of (P_{dc}) and (D_{dc}) (see [10,11,14,19]).

4.2 DC Algorithm (DCA)

Based on local optimality conditions and DC duality, the DCA consists in constructing of two sequences $\{x^k\}$ and $\{y^k\}$ of trial solutions of the primal and dual programs respectively, such that the sequences $\{g(x^k) - h(x^k)\}$ and $\{h^*(y^k) - g^*(y^k)\}$ are decreasing, and $\{x^k\}$ (resp. $\{y^k\}$) converges to a primal feasible solution $\widetilde{x}$ (resp. a dual feasible solution $\widetilde{y}$) satisfying local optimality conditions and

$$\widetilde{x} \in \partial g^*(\widetilde{y}), \quad \widetilde{y} \in \partial h(\widetilde{x}). \tag{18}$$

The sequences $\{x^k\}$ and $\{y^k\}$ are determined in the way that x^{k+1} (resp. y^{k+1}) is a solution to the convex program (P_k) (resp. (D_{k+1})) defined by ($x^0 \in \text{dom } \partial h$ being a given initial point and $y^0 \in \partial h(x^0)$ being chosen)

$$(P_k) \quad \inf\{g(x) - [h(x^k) + \langle x - x^k, y^k\rangle] : x \in \mathrm{X}\},$$

$$(D_{k+1}) \quad \inf\{h^*(y) - [g^*(y^k) + \langle y - y^k, x^{k+1}\rangle] : y \in \mathrm{Y}\}.$$

The DCA has the quite simple interpretation: at the k-th iteration, one replaces in the primal DC program (P_{dc}) the second component h by its affine minorization $h^{(k)}(x) := h(x^k) + \langle x - x^k, y^k\rangle$ defined by a subgradient y^k of h at x^k to give birth to the primal convex program (P_k), the solution of which is nothing but $\partial g^*(y^k)$. Dually, a solution x^{k+1} of (P_k) is then used to define the dual convex program (D_{k+1}) obtained from (D_{dc}) by replacing the second DC component g^* with its affine minorization $(g^*)^{(k)}(y) := g^*(y^k) + \langle y - y^k, x^{k+1}\rangle$ defined by the subgradient x^{k+1} of g^* at y^k. The process is repeated until convergence. DCA performs a double linearization with the help of the subgradients of h and g^* and the DCA then yields the next scheme: (starting from given $x^0 \in \text{dom } \partial h$)

$$y^k \in \partial h(x^k); \quad x^{k+1} \in \partial g^*(y^k), \ \forall k \geq 0. \tag{19}$$

DCA's distinctive feature relies upon the fact that DCA deals with the convex DC components g and h but not with the DC function f itself. DCA is one of the rare algorithms for nonconvex nonsmooth programming. Moreover, a DC function f *has infinitely many DC decompositions which have crucial implications for the qualities* (speed of convergence, robustness, efficiency, globality of computed solutions,...) of DCA. For a given DC program, the choice of *optimal* DC decompositions is still open. Of course, this depends strongly on the very specific structure of the problem being considered. In order to tackle the large-scale setting, one tries in practice to choose g and h such that sequences $\{x^k\}$ and $\{y^k\}$ can be easily calculated, *i.e.,* either they are

in an explicit form or their computations are inexpensive.

We are going to end this section by summarizing the DCA's Convergence Theorem [10]-[25] and also the webpage `http://lita.sciences.univ-metz.fr/~lethi/`.

Theorem 1 (Convergence Theorem of DCA) *([10, 11, 14]) DCA is a descent method without line-search which enjoys the following primal properties (the dual ones can be formulated in a similar way):*

1. *The sequences* $\{g(x^k) - h(x^k)\}$ *and* $\{h^*(y^k) - g^*(y^k)\}$ *are decreasing and*
   - $g(x^{k+1}) - h(x^{k+1}) = g(x^k) - h(x^k)$ *if and only if* $y^k \in \partial g(x^k) \cap \partial h(x^k)$, $y^k \in \partial g(x^{k+1}) \cap \partial h(x^{k+1})$ *and* $[\rho(g, C) + \rho(h, C)]\|x^{k+1} - x^k\| = 0$. *Moreover if* g *or* h *are strictly convex on* C, *then* $x^k = x^{k+1}$. *In such a case DCA terminates at finitely many iterations.*
     Here C *(resp.* D*) denotes a convex set containing the sequence* $\{x^k\}$ *(resp.* $\{y^k\}$*) and* $\rho(g, C)$ *denotes the modulus of strong convexity of* g *on* C *given by:*

$$\rho(g, C) := \sup\{\rho \geq 0 : g - (\rho/2)\|.\|^2 \text{ be convex on } C\}.$$

   - $h^*(y^{k+1}) - g^*(y^{k+1}) = h^*(y^k) - g^*(y^k)$ *if and only if* $x^{k+1} \in \partial g^*(y^k) \cap \partial h^*(y^k)$, $x^{k+1} \in \partial g^*(y^{k+1}) \cap \partial h^*(y^{k+1})$ *and* $[\rho(g^*, D) + \rho(h^*, D)]\|y^{k+1} - y^k\| = 0$. *Moreover if* g^* *or* h^* *are strictly convex on* D, *then* $y^k = y^{k+1}$. *In such a case DCA terminates at finitely many iterations.*

2. *If* $\rho(g, C) + \rho(h, C) > 0$ *(resp.* $\rho(g^*, D) + \rho(h^*, D) > 0$*) then the series* $\{\|x^{k+1} - x^k\|^2\}$ *(resp.* $\{\|y^{k+1} - y^k\|^2\}$*) converges.*
3. *If the optimal value of* P_{dc} *is finite and the infinite sequence* $\{x^k\}$ *and* $\{y^k\}$ *are bounded then every limit point* x^∞ *(resp.* y^∞*) of the sequence* $\{x^k\}$ *(resp.* $\{y^k\}$*) is a critical point of* $g - h$ *(resp.* $h^* - g^*$*).*
4. *DCA has a linear convergence for general DC programs.*
5. *DCA has a finite convergence for polyhedral DC programs.*

Remark 2
1. In practice, DCA can tackle many classes of large-scale nonconvex, smooth/nonsmooth programs and converges quite often to global solutions.
2. The choice of a good initial point is still an open question for DCA, it depends on the specific structure and features of the DC program being considered.
3. Quality of computed solutions by DCA: the lower the corresponding value of the objective is, the better the local algorithm will be.
4. The degree of dependence on initial points: the larger the set (made up of starting points which ensure convergence of the algorithm to a global solution) is, the better the algorithm will be.

5 DCA for solving MVSK problem

In section 3, we have reformulated the MVSK problem as a DC program:

$$\begin{array}{ll} \min & f(\mathbf{x}) = g(\mathbf{x}) - h(\mathbf{x}) \\ s.t. & \mathbf{x} \in \mathbb{R}^n \end{array} \tag{20}$$

where $g(\mathbf{x}) := \frac{\eta}{2}\mathbf{x}^T\mathbf{x} + \chi_{S_{n-1}}(\mathbf{x})$ is a convex function on $\mathbb{R}^n$, and $h(\mathbf{x}) := \frac{\eta}{2}\mathbf{x}^T\mathbf{x} - f(\mathbf{x})$ is a differentiable convex function on S_{n-1} where the constraint set S_{n-1} is the standard $(n-1)$-simplex (a very simple polyhedral convex set).

Note that DCA can handle general nonlinear nonconvex nonsmooth programming within general convex sets (not only polyhedral sets). For polynomial programming, DCA can be used to deal with more general and higher order objective function (not only homogeneous and 4th-order). If we want to solve a DC program with nonconvex sets, for instance, additional nonconvex constraints (such as cardinality constraint, minimal transaction unit constraint) in MVSK model will lead to a mixed integer nonlinear program which is much more difficult to be solved. In such a case, thanks to penalty techniques in DC programming, this problem can be transformed into a DC program and tackled by DCA. Some examples and applications of DC programming with mixed 0-1 variables or mixed integer variables can be found in the papers ([18, 24, 25]).

According to the general framework of DCA, we need to construct two sequences $\{\mathbf{x}^k\}$ and $\{\mathbf{y}^k\}$ by

$$\begin{array}{c} \mathbf{x}^k \longrightarrow \mathbf{y}^k \in \partial h(\mathbf{x}^k) \\ \qquad\quad \swarrow \\ \mathbf{x}^{k+1} \in \partial g^*(\mathbf{y}^k). \end{array} \tag{21}$$

Since $h(\mathbf{x}) := \frac{\eta}{2}\mathbf{x}^T\mathbf{x} - f(\mathbf{x})$ is a continuous and differentiable convex function, for this case, we have explained in section 4 that $\partial h(\mathbf{x}^k)$ reduces to a singleton which is exactly $\{\nabla h(\mathbf{x}^k)\}$. The sequence $\{\mathbf{y}^k\}$ can be computed explicitly via the following equivalence:

$$\begin{aligned} \mathbf{y}^k \in \partial h(\mathbf{x}^k) &= \{\mathbf{y}^k = \nabla h(\mathbf{x}^k)\} \\ &= \{\mathbf{y}^k = \eta\mathbf{x}^k - \nabla f(\mathbf{x}^k)\} \\ &= \{\mathbf{y}^k = \eta\mathbf{x}^k + \alpha\mu - 2\beta\mathbf{V}\mathbf{x} + \tfrac{\gamma\nabla^2 S(\mathbf{x})\mathbf{x}}{2} - \tfrac{\delta\nabla^2 K(\mathbf{x})\mathbf{x}}{3}\}. \end{aligned} \tag{22}$$

For computing $\mathbf{x}^{k+1} \in \partial g^*(\mathbf{y}^k)$, we need to solve the convex quadratic programming subproblem

$$\begin{array}{c} \mathbf{x}^{k+1} \in argmin \;\; \frac{\eta}{2}\mathbf{x}^T\mathbf{x} - \mathbf{x}^T\mathbf{y}^k \\ s.t. \;\; \mathbf{x} \in S_{n-1}. \end{array} \tag{23}$$

This problem (23) is equivalent to the least-square problem:

$$\begin{array}{c} \min \;\; \|\mathbf{x} - \frac{\mathbf{y}^k}{\eta}\|^2 \\ s.t. \;\; \mathbf{x} \in S_{n-1} \end{array} \tag{24}$$

From the geometric viewpoint, the problem (24) is to find a projection point of $\frac{\mathbf{y}^k}{\eta}$ on S_{n-1}.

The problems (23) and (24) are both strictly convex programs, so the optimal solution set is a singleton. Many efficient solution methods could be used. Particularly, when a large-scale problem is encountered, we suggest using direct projection methods, such as the projection method presented in the book of Minoux ([34], pp.250-251).

Another one is the block pivotal principal pivoting algorithm (BPPPA) presented in [35] which is strongly polynomial and easy to be implemented. Given the program:

$$\begin{array}{ll} \min & \frac{1}{2}\mathbf{x}^T\mathbf{x} + \mathbf{q}^T\mathbf{x} \\ s.t. & \mathbf{e}^T\mathbf{x} = p, x_i \geq 0, i = 1, \ldots, n. \end{array} \tag{25}$$

The steps of (BPPPA) method are presented as follows:

Block Pivotal Principal Pivoting Algorithm

Step 0: Let $F = \{1, 2, \ldots, n\}$.

Step 1: Compute $\phi = -\frac{p+\sum_{i\in F} q_i}{|F|}$.

Step 2: Let $H = \{i \in F : q_i + \phi > 0\}$.
If $H = \emptyset$ stop algorithm and $x = (x_i)_{i=1,\ldots,n}$, where $x_i = \begin{cases} 0, & i \notin F; \\ -(q_i + \phi), & i \in F. \end{cases}$
is the optimal solution of the quadratic program (25).
Otherwise, set $F = F - H$ and goto **Step 1**.

Repeating the operations for computing $\{\mathbf{x}^k\}$ and $\{\mathbf{y}^k\}$ until the two sequences converge and/or the objective value of the DC function converges.
Now, we describe the DCA for solving the MVSK problem as follows:

DCA for MVSK Problem

Initialization:
Compute η via the formula (10).
Choose an initial point $\mathbf{x^0} \in S_{n-1}$.
Let ϵ_1, ϵ_2 be sufficiently small positive numbers.
Set iteration number $k = 0$.

Iteration: $k = 0, 1, 2, ...$
- Calculate $\mathbf{y}^k$ via the explicit formula (22).
- Find $\mathbf{x}^{k+1}$ by solving (23) via an explicit method.

Stop Criterions:
If either $||\mathbf{x}^{k+1} - \mathbf{x}^k|| \leq \epsilon_1(1 + ||\mathbf{x}^k||)$
or $|f(\mathbf{x}^{k+1}) - f(\mathbf{x}^k)| \leq \epsilon_2(1 + |f(\mathbf{x}^k)|)$
Then, **STOP** iteration and $\mathbf{x}^{k+1}$ should be a computed solution.
Otherwise, set $k = k + 1$ and repeat `Iteration` step.

Remark 3 This efficient projection method yields explicit computation during each iteration of DCA, and without relying on any third-party quadratic optimization software. Such a simple framework makes it possible to handle large-scale problems.

Theorem 2 (Convergence Theorem of DCA)

– *The sequence* $\{f(\mathbf{x}^k) = g(\mathbf{x}^k) - h(\mathbf{x}^k)\}$ *is decreasing and bounded below (i.e. convergent).*

– *The whole sequence* $\{\mathbf{x}^k\}$ *converges to a KKT point of* (P_{dc})

First, it is clear that there is identity between critical point and KKT point for the DC program (20). Hence it suffices to prove the second property. In virtue of the general convergence of DCA in Section 4, every convergent subsequence of $\{x^k\}$ converges to a KKT point for(20). In fact, according to recent results on convergence analysis of DCA for DC programs with subanalytic data [23], the whole sequence $\{x^k\}$ is convergent.

How to choose a good initial point? The choice of a good initial point is important for DCA, because computational results show that if DCA starts from a good initial point, it will have fast convergence and its computed solution could be coincide with a global optimal one. However, developing a deterministic method for finding good initial points for all DC programs is still an open question. It depends on the structure of the DC decomposition. For our specific DC decomposition to MVSK problem, we propose to find an initial point via the following method.

Firstly, we must understand that the choice of an initial point depends on the investor's preference parameters, since the different choice of parameters might totally change the nature of the problem (1). For instance, if $\gamma = \delta = 0$ and $\beta \neq 0$, the problem (1) becomes a convex quadratic program which is easy to be solved by many existing methods. In this case, there is no need to use DCA (although DCA can handle this case without any problem, and every point of $\mathbb{R}^n$ could be used as an initial point of DCA). However, we prefer using DCA for solving nonconvex programming, i.e. when γ and δ are not both equal to zero. In such a case, we can neglect the skewness and kurtosis in the objective function f in (1), and then an initial point of DCA can be found by solving the following problem:

$$\begin{array}{ll} \min & -\alpha E(\mathbf{x}) + \beta V(\mathbf{x}) \\ s.t. & \sum_{i=1}^{n} x_i = 1, x_i \geq 0, i = 1, \ldots, n. \end{array} \tag{26}$$

Note that when $\beta = 0$, the problem (26) becomes a linear program, when $\beta \neq 0$, it should be a convex quadratic program. Both of them can be efficiently solved by many existing methods. The optimal solution of (26) will be often a good initial point of DCA, since the numerical analysis shows that the value of $S(\mathbf{x})$ and $K(\mathbf{x})$ are often closed to zero (about $0.0001 - 0.01$), while the value of $E(\mathbf{x})$ and $V(\mathbf{x})$ are often more important (about $0.01 - 0.1$).

On the other hand, for the special case when $\alpha = \beta = 0$, we can not use the problem (26) any more, because in this case, the objective function of (26) is identical to zero. One possible method is to approximate f by its second-order Taylor expansion around the point $\bar{\mathbf{x}} := (\frac{1}{n}, \ldots, \frac{1}{n})^T \in \mathbb{R}^n$ (the center of the simplex S_{n-1}). Thus, we can derive an optimization problem as follows:

$$\begin{array}{ll} \min & Q(\mathbf{x}) := \frac{1}{2}\mathbf{x}^T \nabla^2 f(\bar{\mathbf{x}})\mathbf{x} + (\nabla f(\bar{\mathbf{x}}) - \nabla^2 f(\bar{\mathbf{x}})\bar{\mathbf{x}})^T \mathbf{x} \\ s.t. & \sum_{i=1}^{n} x_i = 1, x_i \geq 0, i = 1, \ldots, n. \end{array} \tag{27}$$

Note that the problem (27) is a quadratic program not necessarily convex. We propose to solve the SDP relaxation problem presented as follows:

$$\begin{array}{rl}(\mathbf{x}^*, W^*) \in argmin & \nabla^2 f(\bar{\mathbf{x}}) \bullet W + (\nabla f(\bar{\mathbf{x}}) - \nabla^2 f(\bar{\mathbf{x}})\bar{\mathbf{x}})^T \mathbf{x} \\ s.t. & \sum_{i=1}^n x_i = 1, x_i \geq 0, i = 1, \ldots, n \\ & \begin{bmatrix} 1 & \mathbf{x}^T \\ \mathbf{x} & W \end{bmatrix} \succeq 0 \end{array} \tag{28}$$

where the notation $A \bullet B$ stands for the inner product of the symmetric matrices A and B defined by $A \bullet B = Tr(AB)$. The matrix inequality $\succeq$ is the Löwner partial order on positive semi-definite matrices, $A \succeq 0$ means A is a positive semi-definite matrix. We can solve the SDP problem (28) to find its optimal solution $(\mathbf{x}^*, W^*)$, and $\mathbf{x}^*$ is used as an initial point for DCA.

Global Optimization Method DCA-B&B We can combine DCA with a Branch-and-Bound technique to derive a global DCA-B&B method. While checking the globality of solution computed by DCA, the combination could provide better solution for restarting DCA and improve upper bounds for B&B. As a result, DCA-B&B accelerates the convergence of B&B and it can globally solve large-scale DC programs.

A B&B scheme consists of both branching and bounding procedures. Usual branching procedures involve rectangular/simplicial subdivisions. Due to simplex structure of S_{n-1}, a special subdivision will be used. It can be described as follows:

Given a partition set M which is a $(n-1)$-simplex in S_{n-1} whose vertex set is given by $V(M) = \{v^1, \ldots, v^n\}$, where the vectors $v^1, \ldots, v^n$ are linearly independent. Let $[a, b]$ be the smallest box containing M. We have

$$a_i = \min\{v_i^j : j = 1, \ldots, n\}, b_i = \max\{v_i^j : j = 1, \ldots, n\}, i = 1, \ldots, n.$$

and

$$M = co(V(M)) = \{x \in R^n : x = \sum_{i=1}^n \lambda_i v^i, \sum_{i=1}^n \lambda_i = 1, \lambda_i \geq 0\}$$

For subdivision, we should find the longest edge of the box $[a, b]$ as $[a_i, b_i]$ where

$$i \in \arg\max\{b_j - a_j : j = 1, \ldots, n\}.$$

. This index i will correspond to an edge of the simplex M between two vertices u_1 and u_2 determined by

$$u_1 \in \{v^j \in V(M) : v_i^j = a_i, j = 1, \ldots, n\}$$

$$u_2 \in \{v^j \in V(M) : v_i^j = b_i, j = 1, \ldots, n\}$$

Thus, we can compute the middle point $\frac{u_1+u_2}{2}$ which can be used for constructing a bisection of M into two simplices M_1 and M_2 satisfying:

$$M = M_1 \cup M_2$$

where

$$V(M_1) = (V(M) \setminus \{u_2\}) \cup \{\frac{u_1 + u_2}{2}\}$$

and

$$V(M_2) = (V(M) \setminus \{u_1\}) \cup \{\frac{u_1 + u_2}{2}\}$$

It is clear that M_1 and M_2 are both $(n-1)-$ simplices.

Note that the first partition set M is identical to S_{n-1}, and its corresponding initial box is $[0,1]^n$. The essential difference between the proposed subdivision method and the standard simplicial subdivision is that: the new one does not require computing the length of all edges of M for find its longest edge, and then it is more efficient than the standard simplicial subdivision, especially for large-scale problems.

The bounding procedure consists of constructing upper bounds and lower bounds. For lower bounding, we use the convex relaxation problem to the DC program (20) at each partition set M of S_{n-1} to compute lower bounds of the optimal value ξ of the DC program

$$\xi := \min\{g(x) - h(x) : x \in M\} \tag{29}$$

DC convex relaxation technique Let $co_M(-h)$ denote the convex envelope of the concave function $-h$ on M. Then $g + co_M(-h)$ is a convex minorant of $f = g - h$ on M and the optimal value τ of the following convex program is a lower bound for ξ

$$\tau := \min\{g(x) + co_M(-h)(x) : x \in M\}. \tag{30}$$

This DC relaxation technique developped in our previous works is particuliarly relevant to the DC structure (see the recent paper [25] and references therein) because computing convex envelope of a function is, in general, not numerically tractable.

This technique requires constructing the convex envelope $co_M(-h)$ which is not a difficult task. Indeed it is well-known that for a given concave function $-h$, the function $co_M(-h)$ defined by [25]

$$co_M(-h)(x) = \min\{\sum_{i=1}^{n} \lambda_i(-h(v^i)) : x = \sum_{i=1}^{n} \lambda_i v^i, \sum_{i=1}^{n} \lambda_i = 1, \lambda_i \geq 0\}, \tag{31}$$

is an affine function given by

$$co_M(-h)(x) = \sum_{i=1}^{n} \lambda_i(-h(v^i)), x = \sum_{i=1}^{n} \lambda_i v^i, \sum_{i=1}^{n} \lambda_i = 1, \lambda_i \geq 0,$$

and $co_M(-h)$ agrees with $(-h)$ on $V(M)$. Note that (31) is valid for any bounded polyhedral convex set M, but $co_M(-h)$ then is only polyhedral convex function and we have

$$\begin{aligned} \tau &\leq \min\{g(v^i) + co_M(-h)(v^i) : i = 1, ..., n\} = \min\{g(v^i) - h(v^i) : i = 1, ..., n\} \\ &= \min\{f(v^i) : i = 1, ..., n\} \end{aligned} \tag{32}$$

For computing the lower bound τ related to a $(n-1)-$ subsimplex M of S_{n-1} in our simplicial subdivision B&B, it is useful to reformulate (30) in the sole variable λ of the standard simplex S_{n-1}. Since M is a $(n-1)-$simplex whose affine hull doesn't contain the origin, the $n \times n$ matrix A, whose columns are $v^1, ..., v^n$, is invertible

$$A = \begin{bmatrix} v^1 & \dots & v^n \end{bmatrix}$$

and represents a linear bijective mapping from M onto S_{n-1} :

$$x = A\lambda, \lambda \in S_{n-1}. \tag{33}$$

It follows that (30) can be written as in the following equivalent form (34) (resp. (35)) in the variable x (resp. λ) :

$$\tau := \min\{g(x) - (A^{-T}b)^T x : x \in M\}, \tag{34}$$

where

$$b := \begin{bmatrix} h(v^1) \\ \vdots \\ h(v^n) \end{bmatrix},$$

and

$$\tau := \min\{g(A\lambda) - b^T\lambda : \lambda \in S_{n-1}\}. \tag{35}$$

where

$$g(A\lambda) = \frac{\eta}{2}\lambda^T A^T A\lambda.$$

(34) and (35) are strongly convex quadratic program with the unique solutions x^* and λ^* respectively. They are related by (33). Note that x^* is the projection of $\frac{A^{-T}b}{2}$ on M.

In usual B&B schemes using lower bounding based on (30), lower bounds are increasing (the monotonicity of the lower bound) along with simplicial subdivisions because $co_{M_2}(-h) \leq co_{M_1}(-h)$ if $M_1 \subset M_2$ while upper bounds, being smallest objective values at current feasible solutions, are decreasing. In the combined DCA-B&B,upper bounds are improved by objective values at solution computed by DCA. Moreover, we restart DCA from better feasible solutions generated by B&B to still decrease upper bounds.

The DCA-B&B method could be terminated when the gap between the upper bound and lower bound is small enough (less than a given tolerance ϵ), and then the computed solution should be a global ϵ-optimal solution.

DC concave relaxation in a simplicial B&B DC convex relaxation technique can be applied if computing $co_M(-h)$ is numerically tractable, for example in case M is a bounded polyhedral convex set with known vertex set $V(M)$ whose cardinality $|V(M)|$ is not too large, or in case M is a box $\prod_{i=1}^{n}[a_i, b_i]$ but h is separable: $h(x) = \sum_{i=1}^{n} h_i(x_i)$ ([25] and references therein). DC convex relaxation technique requires minimizing a strongly convex quadratic function on a $(n-1)-$simplex M whose computational cost increases with the dimension n. In our MVSK problem, we can think of the following concave relaxation [12] which consists in replacing the function g by an affine minorant g_0 defined by a subgradient v^0 of g at a point $u^0 \in M$: (we proceed as in DCA by linearizing the DC component g instead of h)

$$g_0(u) := g(u^0) + \langle u - u^0, v^0\rangle, \tag{36}$$

which provides the following lower bound for ξ

$$\zeta(M) := \min\{g_0(u) - h(u) : u \in M\} \tag{37}$$

It is a concave program but easy to solve because the vertex set $V(M)$ is known and $|V(M)| = n$ (not too large):

$$\zeta(M) := \min\{g_0(x) - h(x) : x \in V(M)\} = \min\{g_0(v^i) - h(v^i) : i = 1, ..., n\} \tag{38}$$

Moreover if h is strictly convex then the solution set of (38) is contained in $V(M)$. In MVSK problem the DC decomposition

$$g(x) := \frac{\eta}{2}\|x\|^2,\ h(x) := \frac{\eta}{2}\|x\|^2 - f(x)$$

will give

$$g_0(u) = \eta u^T u^0 - \frac{\eta}{2}\|u^0\|^2$$
$$g_0(u) - h(u) = f(u) - \frac{\eta}{2}\|u - u^0\|^2.$$

The corresponding lower bound then is

$$\zeta(M) := \min\{f(v^i) - \frac{\eta}{2}\|v^i - u^0\|^2 : i = 1, ..., n\} \tag{39}$$

It is clear that ζ depends on u^0 and if we use the same u^0 in the whole B&B scheme, the current lower bound will not change and the corresponding B&B fails to converge. In other words, u^0 should be changed at each simplicial subdivision. For example [12], the following choice of

$$u^0 := \frac{1}{n}\sum_{i=1}^{n} v^i$$

is used.

DC concave relaxation vs DC convex relaxation The above displayed results show that DC convex relaxation is more expansive than DC concave relaxation, but there is, in general, no possible theoretical comparison between their respective convex/concave minorant and consequently no theoretical comparison between their resulting lower bounds. Moreover, unlike DC convex relaxation, concave minorants generated by DC concave relaxation don't enjoy the monotonicity of computed lower bounds, needed to guarantee relevant convergence of the related B&B. To remedy that one can introduce the following natural modification in computing lower bound: If $M_i, i = 1, 2$, are two subsimplices generated from the simplex M as described above, then

$$\zeta(M_i) := \max\{\zeta(M), \zeta(M_i)\},\ i = 1, 2 \tag{40}$$

In practice we observe that DCA-B&B using DC convex relaxation needs less iterations than DCA-B&B using DC concave relaxation, but computing lower bound in the latter is cheaper. Consequently, the first approach (resp. the second one) can be better in some cases.

6 Computational Experiments

We implement our algorithm DCA in MATLAB2007a as well as in C++, and test on a PC equipped with Windows XP, Intel CPU 1.86GHz and 2G RAM.

In this section, we only show the results of the MATLAB version comparing with several other methods and softwares, since most of them are implemented in MATLAB.

1. Gloptipoly: This solver is designed for solving polynomial programming, moment optimization and semi-definite programming under MATLAB environment, which is developed by Didier Henrion and Jean Bernard Lasserre. We use the most recent version Gloptipoly3. This software builds semi-definite relaxation for the input polynomial optimization problem, and then the SDP problem will be solved by the semi-definite programming solver such as SeDuMi and YALMIP. For more information about Gloptipoly, SeDuMi and YALMIP, the reader is referred to ([37–42]).
2. LINGO: A well-known optimization software which is developed by LINDO Systems Inc for solving linear, nonlinear and integer optimization models. We want to compare with its local NLP solver for nonlinear programming with default parameters. In fact, there is also global optimization method in LINGO. More information about LINGO can be found in [43].
3. fmincon: This is a MATLAB's routine which is designed for solving constrained nonlinear optimization. This function is provided in the MATLAB Optimization Toolbox. We can choose two different algorithms for solving nonlinear optimization problems. The first one is *a sequential quadratic programming (SQP) method*. This method solves a quadratic programming subproblem in each iteration. An estimate of the Hessian of the Lagrangian is updated in each iteration using the BFGS formula. The second one is *a trust region method based on an interior-reflective Newton method (Interior Trust Region - ITR)*. Each iteration of this algorithm involves the approximate solution of a linear system using the method of preconditioned conjugate gradients (PCG). For more information about *fmincon* and its algorithms, the reader is referred to the MATLAB's documentation [44] and the references therein.

In order to generate the portfolio test data, we have developed a software in MATLAB which can be used to initialize portfolio data automatically. Firstly, this software can download historical portfolio prices within a given period from the web server of yahoo finance. Then it will compute the return rate for each asset, as well as the moment and co-moment (variance, covariance, skewness, co-skewness, kurtosis and co-kurtosis) needed in MVSK model. The test data presented in this section is a collection of assets from NASDAQ100 index during the period 01/01/2008 - 30/09/2008 with frequency of weeks.

We must emphasize that *the sparsity of the test data is very important*. The difficulty of a polynomial optimization depends not only on the number of variables and constraints, but also on the sparsity of polynomial coefficients. If the polynomial functions have full coefficients, the problem may be intractable even if its number of variables is relatively small. The reason is that the nonzero coefficients will make the polynomial function highly nonlinear and exhibiting multiple local maxima and minima. Thus solving such a nonlinear program is very difficult. In numerical simulation,

Table 2 DCA - Gloptipoly - LINGO - SQP(fmincon) - Interior Trust Region(fmincon)

n	Gloptipoly			DCA		
	Init_T(secs)	Sol_T(secs)	Obj	Init_T(secs)	Sol_T(secs)	Obj
6	0.7176	0.7419	-0.00185373	0.0017	0.6505	-0.00185373
8	1.5687	6.3715	-0.00116820	0.0129	2.7966	-0.00116820
10	3.7302	39.8660	-0.00185374	0.0036	1.7593	-0.00185374
12	7.7149	213.3906	-0.00066544	0.0246	2.8770	-0.00066544
16	28.8922	4061.2763	-0.00083222	0.0516	1.0887	-0.00083222
17	–	–	–	0.0338	1.1289	-0.09063376
18	–	–	–	0.0483	1.2793	0.00083222
20	–	–	–	0.0124	0.2203	-0.09063376
21	–	–	–	0.0124	0.2203	-0.09063376
31	–	–	–	0.0400	0.7497	-0.09063376
40	–	–	–	0.1345	0.1657	-0.00185369
50	–	–	–	0.2445	25.5453	-0.00006260
60	–	–	–	0.1826	0.2654	-0.09063376
70	–	–	–	0.8381	0.9692	-0.09063376
81	–	–	–	1.2036	2.6617	-0.07016813

n	LINGO		SQP		ITR	
	Sol_T(secs)	Obj	Sol_T(secs)	Obj	Sol_T(secs)	Obj
6	2.12	-0.00185373	0.0279	-0.00185369	0.0310	-0.00185369
8	9.31	-0.00116820	0.0828	-0.00116798	0.0921	-0.00116798
10	21.01	-0.00185373	0.1201	-0.00185343	0.0982	-0.00185343
12	3.12	-0.00066540	0.0317	-0.00066382	0.0421	-0.00066382
16	8.19	-0.00083219	0.1282	-0.000812890	0.2531	-0.000812890
17	5.128	-0.09063376	2.7131	-0.09063376	3.231	-0.09063376
18	7.89	0.00083222	0.2788	0.00128002	0.5626	0.00128002
20	15.21	-0.09063376	1.0472	-0.09061380	0.9871	-0.09061380
21	–	–	1.2815	-0.09061380	1.5621	-0.09061379
31	–	–	1.8716	-0.09061380	2.1359	-0.09061380
40	–	–	2.3199	-0.00185368	2.3188	-0.00185369
50	–	–	5.6562	-0.00001056	8.1021	-0.00001056
60	–	–	12.6021	-0.09063376	15.3317	-0.09063376
70	–	–	26.1307	-0.09063376	55.620	-0.09063376
81	–	–	151.8911	-0.07016813	193.131	-0.07016813

we can often observe that, more nonzero elements there are, more computer memory space required, and more computation time needs. However, in our tackled problem, the moment and co-moment coefficients are mostly nonzero elements. This fact restricts the size of solvable problems, and raises a challenge for solving large-scale cases. Let us now look at some experiment results.

In Table 2, the parameters are fixed as $(\alpha, \beta, \gamma) = (0.29, 0.21, 0.4)$, $\epsilon_1 = 1e-5$ and $\epsilon_2 = 1e-8$. We test different size of problems from 6 variables to 81 variables.

For small-size problems with $n \leq 16$: Solution computed by DCA and Gloptipoly are identical. That is not the case for the other existing local solvers LINGO and fmincon (SQP and ITR). Note that SQP and ITR give in most cases the same solutions and they are slightly more rapid than DCA. LINGO provides better solutions than SQP and ITR but it is time-consuming (10-100 times slower than DCA). Of course, being a global algorithm, Gloptipoly is very expensive and DCA is $2-4000$ faster than it.

For problems with $17 \leq n \leq 81$: We have performed DCA, SQP and ITR with the same initial points and found that DCA quite often provides better solutions within

shorter computation time, especially for relatively large-scale problems. It's worth noting the nice advantage of DCA for large-scale problems: its computation time doesn't increase with size of problems.

Furthermore, the solvability of the different solvers can also be observed in Table 2. Gloptipoly can only solve small-size problems with less than 16 variables. An *Out of Memory* error arises for problem with more than 16 variables. The reason is probably due to the full moment coefficients which make the constructions of polynomial functions and SDP relaxations become expensive and intractable. The DCA is available for tests containing less than 82 variables. An *Out of Memory* error also comes up for larger size problems during the initialization process of the moment coefficients. Although the limited size (82) seems not to be so large, but the full nonzero coefficients (about the order n^4) lead to a large size problem and need a lot of memory space (about 385MB for 100 variables). A similar reason restricted the solvability of LINGO to less than 21 variables. Therefore, the non sparsity of the problem is an insurmountable bottleneck for solving large-scale problems. This difficulty is an inherent characterization of the MVSK model. We propose the following two improvement strategies:

The first improvement is using MATLAB MEX programming. We have implemented a C++ MEX interface program incorporated with MATLAB environment for computing the moment and co-moment coefficients, and passing the results to MATLAB. Although, this method does not change the sparsity of the problem, but we can already solve the problems with less than 85 variables in our 2G RAM computer.

The second improvement is to design a more economic storage mode for these full coefficients. As we know, when dealing with higher moments, in generality, each moment can be mathematically represented as a *tensor*. The second moment tensor is the familiar $n \times n$ covariance matrix, whereas the third moment tenser, so-called *skewness tensor* can intuitively be seen as a 3-D cube with height, width, and depth of n. The fourth moment tensor, so-called *kurtosis tensor* can similarly be visualized as a 4-D cube. Thanks to the symmetries of co-variance, co-skewness and co-kurtosis[2], we can neglect the symmetric elements and use sparse matrix structure of MATLAB for saving space. Therefore, we only need to save the different nonzero components, and there are about $\binom{n}{1} + \binom{n+1}{2} + \binom{n+2}{3} + \binom{n+3}{4}$ different nonzero components. This method will save significantly the memory space.

However, in practice, there is still a technical problem due to lack of 3-D and 4-D sparse tensor structure in MATLAB. For overcoming this inconvenience, we have also two methods. The first one is to create a 3-D (resp. 4-D) sparse tensor structure using the cell array in MATLAB. The second one is to *split* the higher moment tensor and joint them together as a big sparse 2-D matrix using the Kronecker tensor product. More precisely about the later one, the 3-D skewness tensor (resp. 4-D kurtosis tensor) with n^3 (resp. n^4) elements could be represented by a $n \times n^2$ (resp. $n \times n^3$) matrix. Thus skewness and kurtosis tensor could be represented as matrices[3] by

$$\mathsf{M_3} = (s_{ijk}) = E[(\mathbf{R} - \boldsymbol{\mu})(\mathbf{R} - \boldsymbol{\mu})^T \otimes (\mathbf{R} - \boldsymbol{\mu})^T]$$

[2] The symmetry relationships are $\sigma_{ij} = \sigma_{ji}$, $s_{ijk} = s_{ikj} = s_{jki} = s_{jik} = s_{kij} = s_{kji}$ and $k_{ijkl} = k_{ijlk} = k_{ikjl} = k_{iklj} = k_{iljk} = k_{ilkj} = k_{jikl} = k_{jilk} = k_{jkil} = k_{jkli} = ...$

[3] The symbol $\otimes$ is referred to as the *Kronecker symbol*.

$$\mathsf{M}_4 = (k_{ijkl}) = E[(\mathbf{R}-\boldsymbol{\mu})(\mathbf{R}-\boldsymbol{\mu})^T \otimes (\mathbf{R}-\boldsymbol{\mu})^T \otimes (\mathbf{R}-\boldsymbol{\mu})^T]$$

where the s_{ijk} and k_{ijkl} are defined by the formulas in section 2. Computational experiments show that, the sparse 2-D matrix using Kronecker product can save more memory space than the sparse 3-D and 4-D tensor structure.

The second improvement changed the sparsity of the problem and saved significantly the memory space, therefore, we can believe that using this method may give new possibilities to tackle the large-scale problems.

In Table 2

- Init_T of Gloptipoly := Time for constructing the polynomial objective function + Time for building the moment SDP problem by invoking **msdp** function.
- "Init_T" of DCA = Time for DC decomposition + Time for computing η via the formula (10) + time for searching a good initial point.

The time-consuming for computing the moment and co-moment coefficients have not been taken into account in Table 2.

Let us point out more details about the performance of DCA. We show an example of 31 assets with parameters $(\alpha, \beta, \gamma) = (0.29, 0.21, 0.4)$, $\epsilon_1 = 1e-5$ and $\epsilon_2 = 1e-8$. The computational result is given in Tables 3 and 4.

Table 3 DCA's Iterations

#Iter	Obj	error_X	error_Obj
1	-0.0027435199	3.1879138770	0.0169478651
2	-0.0173965397	0.1593109323	0.0146530198
3	-0.0312198016	0.1552376217	0.0138232619
4	-0.0441255821	0.1498144942	0.0129057804
5	-0.0561902390	0.1452919463	0.0120646569
6	-0.0674478924	0.1404885214	0.0112576535
7	-0.0776177030	0.1330927259	0.0101698105
8	-0.0855805812	0.1153472690	0.0079628782
9	-0.0906337649	0.0926628538	0.0050531837
10	-0.0906337649	0.0000000000	0.0000000000

Table 4 DCA's Optimal Results

Decomposition parameter(η)	5.809351e-001.
Initialization time (secs)	0.040011
Computation time (secs)	0.749753
Iterations	10
Objective(Minimize)	-0.09063376
Moments after optimization:	
P. Mean	0.348381
P. Variance	0.034540
P. Skewness	2.013382e-003
P. Kurtosis	0.039521

Table 5 DCA vs. DCA-B&B

n	DCA-B&B						DCA		
	Init_T	Sol_T[1]	UB	LB	#DCA[2]	Gap[3] (%)	Init_T	Sol_T	Obj
6	1.26	3.12	-0.00185373	-0.00185373	1	0	0.002	0.65	-0.00185373
8	1.85	8.68	-0.00116820	-0.003162712	1	0.2	0.01	2.79	-0.00116820
10	1.89	52.55	-0.00185374	-0.002852771	1	0.1	0.003	1.76	-0.00185374
12	2.15	83.91	-0.00066544	-0.00066544	1	0	0.024	2.88	-0.00066544
16	2.88	135.40	-0.00083222	-0.001817639	1	0.1	0.051	1.09	-0.00083222
17	2.79	179.21	-0.09063376	-0.09239712	1	0.2	0.033	1.13	-0.09063376
18	2.33	295.88	0.00083222	-0.00277619	1	0.3	0.048	1.28	0.00083222
20	3.15	589.71	-0.09063376	-0.09637820	1	0.6	0.012	0.22	-0.09063376
21	3.78	815.15	-0.09063376	-0.09818287	1	0.8	0.012	0.22	-0.09063376
31	5.32	1685.11	-0.09063376	-0.1028188	1	1.0	0.040	0.75	-0.09063376
40	6.82	1800	-0.00185369	-0.05452321	1	5.3	0.134	0.16	-0.00185369
50	7.11	1800	-0.00006260	-0.1028620	1	10.2	0.244	25.54	-0.00006260
60	8.57	–	-0.09063376	–	1	–	0.182	0.26	-0.09063376
70	9.33	–	-0.09063376	–	1	–	0.838	0.96	-0.09063376
81	11.51	–	-0.07016813	–	1	–	1.203	2.66	-0.07016813

1. The solution time is limited to 1800 seconds.
2. The number of times to start DCA.
3. The relative gap is limited to 1%.

Observing the computing process of DCA, we can find some important features. Firstly, in Table 3, the objective value decreases after each iteration and often decreases more fast at the beginning. Secondly, DCA has a fast convergence (often between $2-30$ iterations). Thirdly, *error_Obj* is often smaller than *error_X* which means that the parameter ϵ_2 is more sensitive than ϵ_1 since the convergence of $\{(g-h)(vecx^k)\}$ is faster than the convergence of $\{\mathbf{x}^k\}$. Therefore, when a higher precision solution is required, we suggest reducing the value of ϵ_2 in order to accelerate its convergence.

For estimating a suitable decomposition parameter η is important. Theoretically, any $\eta \geq \eta^* := max\{\rho(\nabla^2 f(\mathbf{x})) : \mathbf{x} \in S_{n-1}\}$ could derive a DC decomposition. However, it is well-known that the smaller η is, the better the DC decomposition will be. In Table 4, the decomposition parameter η computed via the formula (10) is a small value. Many computational experiments show that the formula (10) can always provide a good estimation of η since its value is often between $0.01 - 1$.

In Table 5, we give some comparison results between DCA and the global optimization approach DCA-B&B. DCA is often executed only once and rarely restarted during the Branch-and-Bound process. This is due to the fact that DCA itself has already found a good upper bound solution without the Branch-and-Bound process. The B&B process is practically used to check the quality of upper bound solutions of DCA. DCA-B&B can solve the problems up to 50 variables. For problems with $31 \leq n \leq 50$ variables, the total solution time for DCA-B&B exceeds 30 minutes. Therefore, we limit the solution time to 1800 seconds and return the best lower bound. Concerning problems with $n > 50$ variables, the computation can not be continued because of an out of memory error. Note that the convergence of DCA-B&B highly depends on the quality of lower bounds.

7 Conclusion and Future work

In this paper, we propose a new efficient DC programming approach for solving a portfolio decision problem when higher moments are taken into account (MVSK model). Some important properties about the polynomial objective function and homogeneous functions, as well as the statements of an appropriate DC decomposition and its related DCA have been investigated. Numerical experiments and computational comparisons show that starting with a good initial point, DCA often converges fast to a better computed solution in a shorter time comparing with some other existing solution methods, especially for relatively large-scale problems. We also study the question on how to find a good initial point for DCA in order to get a high quality solution. Gloptipoly and DCA-B&B allow checking the globality of DCA. It turns out that the solutions computed by DCA are global ones in almost all cases. The DCA confirms again its robustness, efficiency, and globality for nonconvex programming.

Another topic is to modify the MVSK model with additional nonconvex constraints (cardinality constraints, minimal transaction unit constraints). These constraints will transform the continuous optimization problem into a mixed-integer nonlinear program, which is much more difficult to be solved. Concerning polynomial optimization, it is crucial to develop efficient combination of DCA and Gloptipoly in order to globally solve larger size problems. Researches in these directions are currently in progress; their results will be reported subsequently.

Acknowledgements We would like to thanks the two referees and the associate editors for their helpful comments and suggestions which have improved the presentation of the paper.

References

1. Markowitz H.M.: Portfolio Selection, Journal of Finance, Vol 7 No 1, 77-91, (1952)
2. Steinbach M.C.: Markowitz Revisited: Mean-Variance Models in Financial Portfolio Analysis. Society for Industrial and Applied Mathematics, Vol 43 No 1, 31-85, (2001)
3. Zenios S.A., Jobst N.J.: The Tail that Wags the Dog: Integrating Credit Risk in Asset Portfolios. Journal of Risk Finance, Fall 2001, 31-44, (2001)
4. Harvey C.R., Siddique A.: Conditional Skewness in Asset Pricing Tests. The Journal of Finance, Vol 55(3), 1263-1295, (2000)
5. Arditti F.D., Levy H.: Portfolio Efficiency Analysis in Three Moments: The Multiperiod Case. Journal of Finance, American Finance Association, Vol. 30(3), 797-809, (1975)
6. Rockinger M., Jondeau E.: Conditional Volatility, Skewness and Kurtosis: Existence and Persistence and Comovements. Journal of Economic Dynamics and Control, Vol 27, 1699-1737, (2003)
7. Jean W.H.: The extension of portfolio analysis to three or more parameters. Journal of Financial and Quantitative. Analysis 6, 505-515, (1971)
8. Athayde G., Flôres R.: Finding a maximum skewness portfolio. Society for Computational Economics, 271, (2001)
9. Harvey C.R., Liechty J.C., Liechty M.W. and Mueller P.: Portfolio Selection With Higher Moments. Duke University, Working Paper, (2003)
10. Pham Dinh T., Le Thi H.A., DC Programming. Theory, Algorithms, Applications: The State of the Art. First International Workshop on Global Constrained Optimization and Constraint Satisfaction, Nice, October 2-4, (2002)
11. Pham Dinh T., Le Thi H.A., Convex analysis approach to D.C. programming: Theory, Algorithms and Applications. Acta Mathematica Vietnamica, Vol.22 No.1, 289-355, (1997)
12. Pham Dinh T., Le Thi H.A.: DC relaxation techniques for lower bounding in the combined DCA - B&B, Research Report, National Institute for Applied Sciences, Rouen, France, (1996)

13. Pham Dinh T., Le Thi H.A.: A Branch-and-Bound method via DC Optimization Algorithm and Ellipsoidal technique for Box Constrained Nonconvex Quadratic Programming Problems. Journal of Global Optimization Vol. 13, 171-206, (1998)
14. Le Thi H.A., Pham Dinh T.: The DC programming and DCA Revisited with DC Models of Real World Nonconvex Optimization Problems. Annals of Operations Research, Vol. 133, 23-46, (2005)
15. Le Thi H.A., Pham Dinh T.: Large Scale Molecular Optimization From Distance Matrices by a D.C. Optimization Approach. SIAM Journal on Optimization, Vol. 4(1), 77-116, (2003)
16. Le Thi H.A., Pham Dinh T., Le Dung M.: Exact penalty in DC programming. Vietnam Journal of Mathematics, Vol. 27(2), 169-178, (1999)
17. Le Thi H.A., Pham Dinh T.: Solving a class of linearly constrained indefinite quadratic problems by DC Algorithms. Journal of Global Optimization, Vol. 11, 253-285, (1997)
18. Le Thi H.A., Pham Dinh T.: A continuous approach for large-scale constrained quadratic zero-one programming. (In honor of Professor ELSTER, Founder of the Journal Optimization), Optimization Vol. 45(3), 1-28, (2001)
19. Pham Dinh T., Le Thi H.A.: DC optimization algorithms for solving the trust region subproblem, SIAM J.Optimization, Vol. 8, 476-507, (1998)
20. Le Thi H.A.: Solving large scale molecular distance geometry problems by a smoothing technique via the gaussian transform and d.c. programming, Journal of Global Optimization, Vol. 27(4), 375-397, (2003)
21. Le Thi H.A.: An efficient algorithm for globally minimizing a quadratic function under convex quadratic constraints. Mathematical Programming, Ser. A, Vol. 87(3), 401-426, (2000)
22. Le Thi H.A., Pham Dinh T., François A.: Combining DCA and Interior Point Techniques for large-scale Nonconvex Quadratic Programming. Optimization Methods & Software, Vol. 23(4), 609-629, (2008)
23. Le Thi H.A., Pham Dinh T., Huynh Van N., Convergence Analysis of DC Algorithm for DC programming with subanalytic data, preprint, (2009)
24. Niu Y.S., Pham Dinh T.: A DC Programming Approach for Mixed-Integer Linear Programs. Computation and Optimization in Information Systems and Management Sciences, Communications in Computer and Information Science, Springer Berlin Heidelberg, 244-253, (2008)
25. Pham Dinh T., Nguyen Canh N. and Le Thi H.A., An efficient combined DCA and B&B using DC/SDP relaxation for globally solving binary quadratic programs. Journal of Global Optimization, Online first, 12 December, (2009)
26. Rockafellar R.T.: Convex Analysis. Princeton University Press, N.J., (1970)
27. Hiriart Urruty J.B., Lemaréchal C.: Convex Analysis and Minimization Algorithms. Springer, Berlin, Heidelberg (1993)
28. Horst R.: D.C. Optimization: Theory, Methods and Algorithms, in: R. Horst and P.M. Pardalos (eds.), Handbook of Global Optimization, 149-216. Kluwer Academic Publishers, Dordrecht, The Netherlands. (1995)
29. Horst R., Thoai N.V.: DC Programming: Overview. Journal of Optimization Theory and Applications, Vol. 103, 1-43, Springer Netherlands, (1999)
30. Horst R., Pardalos P.M., Thoai N.V.: Introduction to Global Optimization - Second Edition. Kluwer Academic Publishers, Netherlands, (2000)
31. Wolkowicz H., Saigal R., Vandenberghe L.: Handbook of Semidefinite Programming - Theory, Algorithms, and Applications. Kluwer Academic Publishers, USA, (2000)
32. Parpas P., Rustem B.: Global optimization of the scenario generation and portfolio selection problems, Computational Science and Its Applications, Lecture Notes in Computer Science, Vol. 3982, 908-917, (2006)
33. Lai K.K, Yu L., Wang S.Y.: Mean-Variance-Skewness-Kurtosis-based Portfolio Optimization. Proceedings of the First International Multi-Symposiums on Computer and Computational Sciences Volume 2 (IMSCCS'06), 292-297, (2005)
34. Gondran M., Minoux M.: Graphes et algorithmes, 250-251. Eyrolles, Paris (1995)
35. Júdice J.J., Pires F.M.: Solution of Large-Scale Separable Strictly Convex Quadratic Programs on the Simplex. Linear Algebra and its Applications, Vol. 170, 214-220, (1992)
36. Frank J.F., Sergio M.F., Petter N.K.: Financial Modeling of the Equity Market: From CAPM to Cointegration, 131-139. Wiley, USA, (2006)
37. Lasserre J.B.: Global optimization with polynomials and the problem of moments. SIAM J. Optim, Vol. 11(3), 796-817, (2001)

38. Henrion D., Lasserre J.B., Lf̈berg J.: GloptiPoly 3: moments, optimization and semidefinite programming, Version 3.4, 30 september, (2008)
39. Henrion D., Lasserre J.B.: GloptiPoly: global optimization over polynomials with Matlab and SeDuMi. ACM Transactions on Mathematical Software, Vol 29(2), 165-194, (2003)
40. GloptiPoly 3: http://www.laas.fr/~henrion/software/gloptipoly3/
41. SeDuMi 1.2: http://sedumi.ie.lehigh.edu/
42. YALMIP: A Toolbox for Modeling and Optimization in MATLAB, Löfberg J. In Proceedings of the CACSD Conference, Taipei, Taiwan, (2004). http://control.ee.ethz.ch/~joloef/wiki/pmwiki.php
43. LINGO 8.0: LINDO Systems - Optimization Software: Integer Programming, Linear Programming, Nonlinear Programming, Global Optimization. http://www.lindo.com/
44. MATLAB R2007a: Documentation and User Guides. http://www.mathworks.com/

CHAPITRE 7

Programmation quadratique

Sommaire

7.1 Introduction

In this chapter, we will present our preliminary researches on DC programming approaches with SDP representation techniques for solving the general quadratically constrained quadratic programs. Let us consider the following problem :

$$(QCQP) \quad \min \quad f_0(x) := x^T Q_0 x + c_0^T x$$

subject to :

$$f_i(x) = x^T Q_i x + c_i^T x \leq r_i, i = 1, \ldots, L$$

$$x \in [\underline{x}, \bar{x}] \subset \mathbb{R}^n.$$

The variables x is a continuous variable which is restricted in a hyper-rectangular. Moreover, the objective function and constrains are quadratic functions, where all symmetric matrices $Q_i, i = 0, \ldots, L$ can be indefinite, and therefore the problem (QCQP) is a very hard, nonconvex optimization problem.

In fact, solving (QCQP) is an NP-hard problem. To see this, note that there are two constraints $x_1(x_1 - 1) \leq 0$ and $x_1(x_1 - 1) \geq 0$ that are equivalent to the constraint $x_1(x_1 - 1) = 0$, which is in turn equivalent to the constraint $x_1 \in \{0, 1\}$. Hence, any $0 - 1$ integer program can be formulated as a quadratically constrained quadratic program. But it is well known that $0 - 1$ integer programming is NP-hard, so QCQP is also NP-hard. There are many application of this problem, such as all application on $0 - 1$ integer programs (e.g. Max cut problems etc.) and all applications

on quadratic optimization (e.g. portfolio optimization, telecommunication, data mining etc.). Therefore, solving this problem is very important in optimization field. Moreover, since all optimization problems with polynomial objective function and polynomial constraints can be reformulated as a (QCQP) (see [82, 63]). Therefore, if we can efficiently solve the (QCQP), then it is also hopeful to efficiently solve the general polynomial programs via the QCQP reformulation techniques.

In this paper, we will consider solving this problem via DC programming approaches. Firstly, we will represent the (QCQP) as an SDP program with a reverse convex constraint. This problem can be easily reformulated as DC program which will be solved by our proposed DC Algorithm (DCA). As an extension of this method, we also present a general idea for solving the general polynomial optimization. At the end of this chapter, some preliminary numerical results are also reported which show the feasibility and the good performance of our approach.

7.2 DC Reformulation of QCQP

Firstly, let us consider the question that how to represent a quadratic function using SDP formulation (i.e., using the semi-definite matrix to represent a quadratic function). We have already presented many techniques in the Chapter 2. The main idea could be described as follows : Considering a general quadratic function $f_i(x) := x^T Q_i x + c_i^T x$, in order represent this function as SDP formulation, let us introduce an additional matrix variable $W \succeq O$ such that $W = xx^T$. Then we have the following equivalence :

$$f_i(x) = x^T Q_i x + c_i^T x \equiv \begin{cases} f_i(x, W) = Q_i \bullet W + c_i^T x \\ W = xx^T \end{cases} \tag{7.1}$$

where the notation $A \bullet B$ stands for the inner product of the symmetric matrices A and B defined by $A \bullet B = Tr(AB)$. The quadratic function $f_i(x)$ is then transformed as $Q_i \bullet W + c_i^T x$ which is a linear function of the variables x and W with an additional nonconvex quadratic equality constraint $W = xx^T$.

For handling the equality quadratic constraint, we can prove the following theorem :

Theorem 7.1

$$W = xx^T \Longleftrightarrow \begin{cases} \begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O, \\ Tr(W) - \|x\|^2 \leq 0 \end{cases}$$

Proof.

$$W = xx^T \Longleftrightarrow W - xx^T \succeq O \text{ et } W - xx^T \preceq O$$

In virtue of the well-known Schur complement theorem, we can derive

$$W - xx^T \succeq O \Longleftrightarrow \begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O.$$

Moreover, since $W - xx^T \preceq O \Longrightarrow Tr(W - xx^T) = Tr(W) - Tr(xx^T) = Tr(W) - \|x\|^2 \leq 0$. So we prove that

$$W = xx^T \Longrightarrow \begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O \text{ and } Tr(W) - \|x\|^2 \leq 0.$$

Inversely,

$$\begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O \text{ and } Tr(W) - \|x\|^2 \leq 0 \Longleftrightarrow W - xx^T \succeq O \text{ and } Tr(W - xx^T) \leq 0.$$

Since the inequality $W - xx^T \succeq O$ implies that all eigenvalues of the matrix $W - xx^T$ are non negatives, and $Tr(W - xx^T) \leq 0$ means that the sum of all eigenvalues of the matrix $W - xx^T$ is not positive. These two conditions are both satisfied if and only if all eigenvalues equal to zero which means $W = xx^T$. □

Finally, we have the following equivalent formulation for the general quadratic function as :

$$f_i(x) = x^T Q_i x + c_i^T x \equiv \begin{cases} f_i(x, W) = Q_i \bullet W + c_i^T x \\ \begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O \\ Tr(W) - \|x\|^2 \leq 0 \end{cases} \tag{7.2}$$

Applying this method, we can reformulate the (QCQP) as the following equivalent problem :

$$(P) \quad \alpha := \min \quad f(x, W) := Q_0 \bullet W + c_0^T x$$

subject to :

$$Q_i \bullet W + c_i^T x \leq r_i, i = 1, \ldots, L$$

$$\begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O$$

$$Tr(W) - \|x\|^2 \leq 0$$

$$(x, W) \in [\underline{x}, \bar{x}] \times [\underline{W}, \bar{W}]$$

All variables x and W of the problem (P) are bounded. Because $x \in [\underline{x}, \bar{x}]$ is bounded, and w_{ij}'s are defined by $w_{ij} := x_i x_j$. So W is also a bounded variable and we can introduce its bounds as

$$\underline{w}_{ij} \leq w_{ij} \leq \bar{w}_{ij} (i, j = 1, \ldots, n) \tag{7.3}$$

where

$$\underline{w}_{ij} = \min\{\underline{x}_i \underline{x}_j, \underline{x}_i \bar{x}_j, \bar{x}_i \underline{x}_j, \bar{x}_i \bar{x}_j\},$$

$$\bar{w}_{ij} = \max\{\underline{x}_i \underline{x}_j, \underline{x}_i \bar{x}_j, \bar{x}_i \underline{x}_j, \bar{x}_i \bar{x}_j\}.$$

Note that the problem (P) should be a SDP program without the reverse convex constraint $Tr(W) - \|x\|^2 \leq 0$.

Let's denote $p(x, W) := Tr(W) - \|x\|^2$, it is obvious to prove the following proposition

Proposition 7.2
- *The fonction $p(x, W)$ is a separable quadratic fonction.*
- $p(x, W) \geq 0$ *when* $\begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O$.
- *Any feasible solution (x, W) of (P) must satisfy $p(x, W) = 0$.*

Thanks to the penalty techniques and the special properties of p, we can introduce the following penalized problem :

$$(P_t) \quad \alpha(t) := \min \quad f_t(x, W) := Q_0 \bullet W + c_0^T x + tp(x, W)$$

subject to :

$$Q_i \bullet W + c_i^T x \leq r_i, i = 1, \ldots, L$$

$$\begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O$$

$$(x, W) \in [\underline{x}, \bar{x}] \times [\underline{W}, \bar{W}]$$

where t is a positive penalty parameter. (P_t) is a concave minimization problem over a compact convex set, it is a classical DC program which can be handled very well by the DC Algorithm (DCA).

Note that since the convex constraint $\begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O$ is not polyhedral, so we are not sure whether we have the exact penalization. However, since all variables of (P) and (P_t) are bounded, and their objective functions are also bounded in the set of constraints. Moreover, the function p is nonnegative under the constraints of (P) and (P_t) (see propositions 7.2). According to the general result of penalty techniques [79], when $t \to +\infty$, the sequence of optimal values $\{\alpha(t)\}$ converges to α. Moreover, for a given large number t, the minimizer (x^*, W^*) of (P_t) must be found in a region where $p(x^*, W^*)$ is closed or equal to zero. So it is not difficult to understand that, when t is large enough, a minimizer (x^*, W^*) of (P_t) should be at least an approximate solution of (P) with $p(x^*, W^*) \approx 0$. How to estimate a suitable value of t is still an open question. In practice, t is often fixed as a relatively large positive real number and we solve the problem (P_t) instead of (P).

Let's denote
$\mathcal{C} := \{(x, W) \in [\underline{x}, \bar{x}] \times [\underline{W}, \bar{W}] : Q_i \bullet W + c_i^T x \leq r_i, i = 1, \ldots, L; \begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O\}$ a

nonempty compact convex set. We present a DC formulation of (P_t) as :

$$(P_{dc}) \quad \min\{f_t(x,W) := g(x,W) - h_t(x,W) : (x,W) \in \mathbb{R}^n \times S^n\}$$

where $g(x,W) := \chi_{\mathcal{C}}(x,W)$ is a convex function ($\chi_{\mathcal{C}}(x,W)$ denotes the indicator function defined by $\chi_{\mathcal{C}}(x,W) := 0$ if $(x,W) \in \mathcal{C}$; $+\infty$ otherwise).
$h_t(x,W) := t\|x\|^2 - W \bullet (Q_0 + tI) - c_0^T x$ is a convex quadratic function. the matrix I is the unit matrix of dimension $n \times n$.
In the next section, we will focus on how to solve this DC program via DCA.

7.3 DCA for solving $(QCQP)$

In the previous section, we have investigated how to reformulate the (QCQP) as a DC program (P_{dc}). For convenience discourse, we rewrite the program (P_dc) as follows :

$$(P_{dc}) \quad \min\{f_t(x,W) := g(x,W) - h_t(x,W) : (x,W) \in \mathbb{R}^n \times S^n\}$$

where

$$g(x,W) := \chi_{\mathcal{C}}(x,W)$$

is a convex function and

$$h(x,W) := t\|x\|^2 - W \bullet (Q_0 + tI) - c_0^T x$$

is a convex quadratic function.

According to the general framework of DCA, we should construct two sequences $\{(x^k, W^k)\}$ and $\{(y^k, Z^k)\}$ via the following scheme :

$$\begin{array}{c} (x^k, W^k) \longrightarrow (y^k, Z^k) \in \partial h(x^k, W^k) \\ \qquad\qquad \swarrow \\ (x^{k+1}, W^{k+1}) \in \partial g^*(y^k, Z^k). \end{array} \tag{7.4}$$

Since $h(x,W)$ is a continuous and differentiable convex quadratic function, $\partial h(x^k, W^k)$ reduces to a singleton $\{\nabla h(x^k, W^k)\}$ which could be computed explicitly :

$$(y^k, Z^k) \;\in\; \partial h(x^k, W^k) = \{y^k = 2tx^k - c_0; Z^k = Q_0 - tI\}. \tag{7.5}$$

For computing $(x^{k+1}, W^{k+1}) \in \partial g^*(y^k, Z^k)$, we can solve the SDP program :

$$(x^{k+1}, W^{k+1}) \in \arg\min\{-x^T y^k - Z^k \bullet W : (x,W) \in \mathcal{C}\}. \tag{7.6}$$

Finally, DCA yields the iterative scheme :

$$(x^{k+1}, W^{k+1}) \in argmin\{-x^T(2tx^k - c_0) - W \bullet (Q_0 - tI) : (x,W) \in \mathcal{C}\}. \tag{7.7}$$

The program (7.7) is a SDP program which can be efficiently solved by many SDP "solvers" such as SDPA [83], SeDuMi[101], CSDP[73], DSDP[74], and Yalmip [102] etc.

For terminating the algorithm, we can use the general stopping criterion of DCA, i.e., one of the following conditions should be satisfied : $|f(x^k, W^k) - f(x^{k-1}, W^{k-1})| \leq \varepsilon_1$ or/and $\|x^k - x^{k-1}\|_2 + \|W^k - W^{k-1}\|_1 \leq \varepsilon_2$ (where $\varepsilon_1, \varepsilon_2$ are given small positive numbers, e.g., $\varepsilon_1 = 10^{-6}, \varepsilon_2 = 10^{-5}$).

Now, we can describe DCA for solving the problem (P_{dc}), i.e., (QCQP) as :

DCA for (QCQP)

Initialization :
Choose an initial point $(x^0, W^0) \in \mathbb{R}^n \times S^n$.
Let ε_1, ε_2 be sufficiently small positive numbers.
Let t be relatively large positive number.
Set iteration number $k = 0$.

Iteration : $k = 0, 1, 2, ...$
Solve the SDP program (7.7) for computing (x^{k+1}, W^{k+1}) from (x^k, W^k).

stopping criterions :
If $|f(x^k, W^k) - f(x^{k-1}, W^{k-1})| \leq \varepsilon_1$ or/and $\|x^k - x^{k-1}\|_2 + \|W^k - W^{k-1}\|_1 \leq \varepsilon_2$ then
Stop algorithm and (x^{k+1}, W^{k+1}) is the computed solution.
Otherwise, let $k = k + 1$ and repeat **Iteration**.

Theorem 7.3 (Convergence Theorem of DCA)
- *DCA can generate a sequence* $\{(x^k, W^k)\}$ *such that the sequence* $\{f(x^k, W^k)\}$ *is decreasing and bounded below (i.e., convergent).*
- *The sequence* $\{(x^k, W^k)\}$ *will converge to a KKT point of* (P_{dc}).

The proof of this theorem is based on the general convergence theorem of DCA which could be found in the papers [29, 23, 32].

7.3.1 Initial point strategy for DCA

The DCA algorithm proposed above need an initial point $(x^0, W^0) \in \mathbb{R}^n \times S^n$. In general, the choice of a good initial point is important for DCA since it will affect the convergence rate, as well as the quality of the numerical solution. In this problem, we suggest relaxing the reverse convex constraint $Tr(W) - \|x\|^2 \leq 0$ of (P) as a linear constraint :

$$Tr(W) + \langle \alpha, x \rangle + \beta \leq 0$$

where α and β are defined as

$$\alpha = -(\underline{x} + \bar{x})$$

$$\beta = \langle \underline{x}, \bar{x} \rangle$$

Since x is restricted in the box $[\underline{x}, \bar{x}]$, then the following inequality holds :

$$Tr(W) - (\underline{x} + \bar{x})^T x + \langle \underline{x}, \bar{x} \rangle \leq Tr(W) - \|x\|^2, \forall x \in [\underline{x}, \bar{x}].$$

Therefore, we have a linear relaxation to the reverse convex constraint.

Finally, we derive the following SDP relaxation problem of (P) :

$$(RP) \quad \min \quad f(x, W) := Q_0 \bullet W + c_0^T x$$

subject to :

$$Q_i \bullet W + c_i^T x \leq r_i, i = 1, \ldots, L$$

$$\begin{bmatrix} 1 & x^T \\ x & W \end{bmatrix} \succeq O$$

$$Tr(W) - (\underline{x} + \bar{x})^T x + \langle \underline{x}, \bar{x} \rangle \leq 0$$

$$x \in [\underline{x}, \bar{x}] \subset \mathbb{R}^n.$$

The optimal solution of (RP) is suggest to be used as an initial point for DCA.

7.3.2 Improvement strategy for large-scale problems

In the procedure of DCA, we have to solve a SDP subproblem during each iteration. However, solving a large-scale SDP problem is also expensive in computation, excepte for some very special structure problems such as sparse structure, block structure etc. We have tested several well-known SDP solvers such as SDPA, SeDuMi, PENBMI, CSDP, DSDP, SDPT etc. In our knowledge, they are best and powerful SDP solvers at the present time. The numerical tests show that for solving a randomly generated sparse structured SDP program of 500 variables and 300 linear constraints needs about $15 - 20$ seconds.

The SDP subproblem requires in DCA possessing $\frac{n(n+3)}{2}$ variables, $L + 1$ linear constraints (reformulated from quadratic constraints), box constraints and a rank-one LMI constraint of order $(n+1) \times (n+1)$. When n is bigger than 100, the SDP problem has more than 5150 variables, solving such a large-scale SDP is very expensive. In order to overcome this difficulty for reducing the total computational time, we propose a *partial solution strategy*, i.e., we don't solve entirely a SDP subproblem of DCA but terminate the SDP's solution procedure when some conditions satisfying. More precisely, the objective for solving a SDP program during each iteration of DCA is to search a better feasible solution x^{k+1} than the current feasible solution x^k such that $(g - h)(x^{k+1}) < (g - h)(x^k)$. Clearly, the minimizer of SDP subproblem is the best

solution can be found in that iteration, but it could be very expensive in computation to find that solution for large-scale problem. Therefore, we propose stopping SDP solution procedure if the first better feasible solution is found within a reasonable time. Using this strategy, the computation time for each iteration of DCA is less expensive, thus it's a promising method to accelerate the convergence for large-scale DC programs.

7.4 Polynomial programs

A polynomial program aims at minimizing a polynomial over a set defined by polynomial inequalities as :

$$\begin{array}{lll} (PP) & \min & p_0(x) \\ & s.t. & p_i(x) \leq 0, i = 1, \ldots, m. \end{array} \tag{7.8}$$

The polynomial program (PP) seems much more general than nonconvex QCQP, while all PPs can be transformed into QCQPs (see [82, 63]). The reason is described as follows : Firstly, we can reduce the maximum degree of a polynomial equation by adding variables. For example, we can turn the constraint

$$x_i^{2n} + (\ldots) \leq 0$$

into

$$y^n + (\ldots) \leq 0, \quad y = x_i^2.$$

We have reduced the degree of the original polynomial inequality by introducing a new variable and a quadratic equality constraint. This method is also available to product terms as :

$$x_1 x_2 x_3 + (\ldots) \leq 0$$

is equivalent to

$$y x_3 + (\ldots) \leq 0, \quad y = x_1 x_2.$$

in the above example, we have replaced a product of three variables by a quadratic term and an additional quadratic equality constraint. By applying these transformation iteratively, we can transforme all higher degree monomials (super than the degree 2) into quadratic terms with some additional quadratic equality constraints. Thus the original polynomial program is turned into a nonconvex QCQP with additional variables.

Example 1 Let's show a specific example :

$$\begin{array}{lll} \min & x_1^5 + x_2^4 + x_1 x_2^2 x_3 + x_3^2 \\ s.t. & x_1^3 + x_1 x_2 x_3 + x_2^2 - 2 \leq 0. \end{array} \tag{7.9}$$

There are 3 *variables* $x_1, x_2, x_3 \in \mathbb{R}$ *in this polynomial program. We will introduce additional variables and quadratic equality constraints to all monomials of degree*

higher than 2 *as follows :*
Replacing x_1^5 *by* x_1x_4 *with* $x_4 = x_5^2, x_5 = x_1^2$.
Replacing x_2^4 *by* x_6^2 *with* $x_6 = x_2^2$.
Replacing $x_1x_2^2x_3$ *by* x_1x_7 *with* $x_7 = x_3x_6$.
Replacing x_1^3 *by* x_1x_5.
Replacing $x_1x_2x_3$ *by* x_1x_8 *with* $x_8 = x_2x_3$.

The problem is then becomes :

$$\begin{array}{ll} \min & x_1x_4 + x_6^2 + x_1x_7 + x_3^2 \\ s.t. & x_1x_5 + x_1x_8 + x_2^2 - 2 \leq 0 \\ & x_4 = x_5^2 \\ & x_5 = x_1^2 \\ & x_6 = x_2^2 \\ & x_7 = x_3x_6 \\ & x_8 = x_2x_3, \end{array} \tag{7.10}$$

which is a nonconvex QCQP with variables $x_1, \ldots, x_8 \in \mathbb{R}$.

We can solve this QCQP (7.10) via our proposed DCA to find its optimal solution $(x_1^*, \ldots, x_8^*)$ in which the part (x_1^*, x_2^*, x_3^*) is also an optimum of the problem (7.9).

The software implementation and numerical simulation of the proposed approaches are under development. Deeper and wider researches on these topics will be reported in the recent future.

7.5 Conclusion

In this paper, we present our preliminary works on DC programming approach for solving the nonconvex Quadratically Constrained Quadratic Program (QCQP), and extend this approach for handling the general Polynomial Program (PP). The SDP representation technique and penalty technique is investigated in order to reformulate a QCQP (resp. PP) as a DC program, and then an efficient DC Algorithm (DCA) is applied for numerical solutions. The initialization strategy for DCA is also studied.

The proposed DC formulation technique can be used in our further researches on Mixed Integer Quadratically Constrained Quadratic Program, as well as Mixed Integer Polynomial Program (MIPP). Moreover, a hybrid global optimization method combining DCA with Branch-and-bound for global solving (QCQP) and (PP) is also an interesting topic. Researches on these problems will be reported subsequently.

Quatrième partie

Programmation sous contraintes des matrices semi-définies

Chapitre 8

Problème de Réalisabilité du BMI et QMI

manuscript No.
(will be inserted by the editor)

New DC Programming Approaches for BMI and QMI Feasibility Problems

Yi-Shuai NIU · Pham Dinh Tao

Received: date / Accepted: date

Abstract We propose some new DC programming approaches for solving the Bilinear Matrix Inequality (BMI) Feasibility Problems and the Quadratic Matrix Inequality (QMI) Feasibility Problems. They are both important problems in the field of robust control and system theory. The inherent difficulty lies in the nonconvex set of feasible solutions. We will firstly reformulate these problems as a DC program (minimization of a concave function over a convex set), and then solved by an efficient DC programming approaches (DCA). Each iteration of DCA requires solving a semidefinite program (SDP). A hybrid method combining DCA with an adaptive Branch-and-Bound (DCA-B&B) is proposed for guaranteeing the feasibility of the BMI and QMI. We suggest a "partial solution" concept for SDP subproblems during iterations of DCA in order to improve DCA for handling large-scale problems. Numerical simulations and comparisons between the proposed approaches and PENBMI are also reported.

Keywords BMI · QMI · Feasibility Problem · DC Programming · DCA · SDP · Branch-and-Bound · DCA-B&B · PENBMI

1 Introduction

The optimization problem with *Bilinear Matrix Inequality* (BMI) constraints is considered as the central problem in the filed of robust control. A wide range of difficult control synthesis problems, such as the fixed order $\mathcal{H}^\infty$ control, the μ/K_m-synthesis, the decentralized control, and the robust gain-scheduling can be reduced to problems involving BMIs [2,3]. Given the symmetric matrices $F_{ij} \in \mathcal{S}^k (i = 0, \ldots, n; j = 0, \ldots, m)$,

Yi-Shuai NIU
Laboratory of Mathematics
National Institute for Applied Sciences - Rouen, France
E-mail: niuyishuai@hotmail.com

Pham Dinh Tao
Laboratory of Mathematics
National Institute for Applied Sciences - Rouen, France
E-mail: pham@insa-rouen.fr

where $\mathcal{S}^k$ denotes the space of real symmetric $k \times k$ matrices. We call the *Bilinear Matrix Inequality* (BMI) an inequality for which the biaffine combination of the given matrices is a negative (or positive) semi-definite (or definite) matrix, mathematically defined as

$$F(\mathbf{x}, \mathbf{y}) := F_{00} + \sum_{i=1}^{n} x_i F_{i0} + \sum_{j=1}^{m} y_j F_{0j} + \sum_{i=1}^{n} \sum_{j=1}^{m} x_i y_j F_{ij} \preceq 0 \tag{1}$$

where $F(x, y) \in \mathcal{S}^k$, $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^m$. The inequality $A \preceq 0$ means the matrix A is negative semi-definite. The BMI could be considered as the generalization of bilinear inequality constraint (including quadratic inequalities) since once F_{ij}'s are diagonals, it becomes a set of bilinear inequality constraints. Therefore, any bilinear and quadratic constraint can be represented as a BMI [4]. Furthermore, it is well-known that BMIs have a biconvexity structure since the BMI constraint becomes convex *Linear Matrix Inequality* (LMI) constraint when the variable $\mathbf{x}$ or $\mathbf{y}$ is fixed. Generally, a LMI is defined as $F(x) := F_0 + \sum_{i=1}^{n} x_i F_i \preceq 0$. It is also known that the LMI is a special case of the BMI. Since the LMI is equivalent to the *Semi-Definite Program* (SDP), the BMI could also be considered as a generalization of the SDP. Therefore, the research value of the BMI is tremendous and not only important in the context of robust control, but also in the development of the mathematical programming theorems and algorithms.

In this paper, we will consider the *BMI Feasibility Problem* (BMIFP). The objective of the BMIFP is to find a point $(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^n \times \mathbb{R}^m$ which satisfies the matrix inequality (1). In fact, we often observe in control theory that the vectors $\mathbf{x}$ and $\mathbf{y}$ are bounded variables, i.e. $(\mathbf{x}, \mathbf{y})$ should be in a hyper-rectangle $\mathcal{H} := \{(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^n \times \mathbb{R}^m : \underline{x} \leq x \leq \bar{x}, \underline{y} \leq y \leq \bar{y}\}$, where $\underline{x}, \bar{x} \in \mathbb{R}^n$ and $\underline{y}, \bar{y} \in \mathbb{R}^m$ are given constant vectors. Some problems arising from the robust control theory are formulated in a more general context as an optimization problem with a linear objective function with BMIs constraints. However, even to solve the BMIFPs are already complex, and it was known to be classified as a NP-hard problem [5]

Some previous researches to BMIs are briefly described to give a view about what has been already done. Of course, it does not cover all works in this topic. The first paper that formally introduced the BMI in the control theory is probably due to Safonov et al. [6] in 1994. They have shown that the BMIFP is equivalent to checking whether the diameter of a certain convex set is greater than two. And this is equivalent to a maximization of a convex function via a convex set which is a NP-hard problem. Later, Goh et al. [2] presented the first implemented algorithm based on convex relaxation and Branch-and-Bound (B&B) scheme for finding a global solution of a particular case of BMIFP. The same authors also proposed algorithms to approximate local optima. Some other authors such as VanAntwerp [7], Fujioka [8] improved the lower bound of the B&B algorithm through a better convex relaxation. Kojima et al. [9] proposed a Branch and Cut algorithm and improved the B&B algorithm of Goh [2] by applying a better convex relaxation of the BMI Eigenvalue Problem (BMIEP). Kawanish et al. [10] and Takano et al. [11] presented some results for reducing the feasible region of the problem using information of local optimal solutions. Tuan et al. [1] had pointed out that BMIFP belongs to the class of *DC (difference of two convex functions)* optimization problems. DC reformulation of BMIFP and B&B algorithm have been investigated in that paper. Later, a Lagrangian dual global optimization algorithm is proposed by

the same author in [16]. A sequence of concave minimization problems and DC programs were employed by Liu and Papavassilopoulos [12] on the same problem of Tuan. Mesbahi and Papavassilopoulos [13] have studied the theoretical aspects of the BMI and established equivalent formulations of cone programming. More recently, the generalized Benders decomposition approach has been extended to BMIFPs by Floudas, Beran et al. [14, 15].

This paper proposes a *DC programming approach* (DCA), as well as a hybrid method combined DCA with a Branch-and-Bound algorithm for solving the general BMIFP. As we've known, the BMI can be reformulated as DC program (concave minimization problem over a convex set). Our proposed method DCA is efficient to solve such DC programs. More specifically, DCA is an iterative method consists of solving a SDP problem at each iteration. The fast convergence rate of DCA raises interest in the combination method of DCA and Branch-and-Bound (DCA-B&B) for solving the concave minimization problem. The particular features of BMIFP and its DC reformulation make it possible to stop the algorithm DCA and DCA-B&B before finding a local and/or global optimal solution. This feature will accelerate the convergence of DCA and DCA-B&B, especially efficient for feasible BMIFP. Some numerical experiments comparing DCA, DCA-B&B and the commercial BMI solver - PENBMI [38] are also reported.

2 Problem Reformulation

In this section, we will firstly present an equivalent formulation of BMIFP which is called *Quadratic Matrix Inequality Feasibility Problem* (QMIFP) defines as follows: Given the symmetric matrices $Q_i \in \mathcal{S}^k (i = 0, \ldots, N)$ and $Q_{ij} \in \mathcal{S}^k (i, j = 1, \ldots, N)$. Finding a point $\mathbf{x}$ in a hyper-rectangle $\mathcal{H} := \{\mathbf{x} \in \mathbb{R}^N : \underline{\mathbf{x}} \leq \mathbf{x} \leq \bar{\mathbf{x}}\}$ who satisfies the following Quadratic Matrix Inequality

$$Q(\mathbf{x}) := Q_0 + \sum_{i=1}^{N} x_i Q_i + \sum_{i,j=1}^{N} x_i x_j Q_{ij} \preceq 0. \tag{2}$$

It is known that the BMIFP (1) can be easily transformed to a QMIFP (2) formulation described in the following proposition:

Proposition 1 *Let $N := n + m$. The BMIFP (1) is equivalent to the QMIFP (2) with*

$$x_i := \begin{cases} x_i & i=1,\ldots,n \\ y_{(i-m)} & i=n+1,\ldots,N; \end{cases}$$

$$Q_0 := F_{00}; Q_i := \begin{cases} F_{i0} & i=1,\ldots,n; \\ F_{0(i-n)} & i=n+1,\ldots,N. \end{cases}$$

$$Q_{ij} := \begin{cases} F_{i(j-n)} & i=1,\ldots,n; j=n+1,\ldots,N \\ F_{(i-m)j} & i=m+1,\ldots,N; j=1,\ldots,m \\ 0 & \textit{otherwise.} \end{cases}$$

$$\underline{x}_i := \begin{cases} \underline{x}_i & i=1,\ldots,n \\ \underline{y}_{(i-m)} & i=n+1,\ldots,N; \end{cases} \textit{ and } \bar{x}_i := \begin{cases} \bar{x}_i & i=1,\ldots,n \\ \bar{y}_{(i-m)} & i=n+1,\ldots,N. \end{cases}$$

The QMIFP and the BMIFP are equivalent in the sense that they have the same feasibility (both to be feasible or infeasible). It is easy to verify that $(\mathbf{x}, \mathbf{y})$ is feasible to BMIFP (1) if and only if $\hat{x}$ defined as $(\mathbf{x}, \mathbf{y})$ is also feasible to the QMIFP (2). Solving such a QMIFP is also a NP-hard problem. However, the advantage of QMIFP lies in the same index rang of i and j between 1 and N. This feature makes it easy to reformulate the QMIFP as a DC program. Once a DC formulation is given, we can solve it via the proposed DC programming approach. In next section, we will give the DC reformulation of the QMIFP (a concave minimization over compact convex set).

3 DC Reformulation of QMIFP

This section will focus on the DC reformulation for the general QMIFPs. The general QMIFP (2) could be reformulated as a concave minimization problem over compact convex set (LMIs constraints and bounds constraints) which has been presented in the paper of H.D.Tuan [1]. We have the following proposition.

Proposition 2 *Let us define $W := (w_{ij}) := \mathbf{x}\mathbf{x}^T$, where $\mathbf{x}^T$ denotes the transpose of the vector $\mathbf{x}$. Considering the matrix $Q(\mathbf{x})$ of the QMIFP (2) in which we can replace all bilinear terms $x_i x_j$ with new variables w_{ij} to obtain the following formulation:*

$$Q_L(\mathbf{x}, W) := Q_0 + \sum_{i=1}^{N} x_i Q_i + \sum_{i,j=1}^{N} w_{ij} Q_{ij} \preceq 0 \tag{3}$$

where

$$(\mathbf{x}, W) \in \{(\mathbf{x}, W) \in \mathbb{R}^N \times S^N : \mathbf{x} \in \mathcal{H}, W = (w_{ij}) = \mathbf{x}\mathbf{x}^T\}.$$

The formulation (3) is equivalent to the general QMIFP formulation (2). More specifically, $\mathbf{x}^*$ is a feasible solution of (2) if and only if $(\mathbf{x}^*, W^* = \mathbf{x}^*\mathbf{x}^{*T})$ is a feasible solution of (3).

In the formulation (2), the difficulty lies in the nonconvex QMI constraint, while in the later formulation (3), the quadratic matrix function $Q(\mathbf{x})$ becomes an affine matrix function $Q_L(\mathbf{x}, W)$, so that the nonconvex QMI constraint has been relaxed to a convex LMI constraint. However, a nonlinear nonconvex equality constraint $W = \mathbf{x}\mathbf{x}^T$ has been introduced in the later one. The Lemma 1 will give us an equivalent formulation to this equality constraint.

Lemma 1 *Let $W \in S_+^N$ defined as $W := \mathbf{x}\mathbf{x}^T$. Then the constraint $W = \mathbf{x}\mathbf{x}^T$ is equivalent to*

$$\begin{bmatrix} W & x \\ x^T & 1 \end{bmatrix} \succeq 0 \tag{4}$$

$$Trace(W - \mathbf{x}\mathbf{x}^T) = Trace(W) - \sum_{i=1}^{N} x_i^2 \leq 0 \tag{5}$$

The proof of lemma 1 can be found in the paper of H.D.Tuan (see lemma 5 of [1]). This lemma helps us to represent the nonlinear equality constraint $W := \mathbf{x}\mathbf{x}^T$ as a LMI constraint (4) and a reverse convex constraint (5).

Therefore, we obtain an equivalent reformulation to the general QMIFP (2) as:

$$Q_0 + \sum_{i=1}^{N} x_i Q_i + \sum_{i,j=1}^{N} w_{ij} Q_{ij} \preceq 0; \begin{bmatrix} W & \mathbf{x} \\ \mathbf{x}^T & 1 \end{bmatrix} \succeq 0; Trace(W) - \sum_{i=1}^{N} x_i^2 \leq 0; \mathbf{x} \in \mathcal{H}. \quad (6)$$

The system (6) could be represented as intersection of the convex set $\mathcal{C} := \{(\mathbf{x}, W) : Q_0 + \sum_{i=1}^{N} x_i Q_i + \sum_{i,j=1}^{N} w_{ij} Q_{ij} \preceq 0; \begin{bmatrix} W & \mathbf{x} \\ \mathbf{x}^T & 1 \end{bmatrix} \succeq 0; \mathbf{x} \in \mathcal{H}.\}$ and the nonconvex set (reverse convex set) $\mathcal{RC} := \{(\mathbf{x}, W) : Trace(W) - \sum_{i=1}^{N} x_i^2 \leq 0\}$. We have the following theorem.

Theorem 1 *The QMIFP (6) is feasible if and only if there exists a point $(\mathbf{x}, W)$ in $\mathcal{C} \neq \emptyset$ and $Trace(W) - \sum_{i=1}^{N} x_i^2 = 0$. Otherwise, the QMIFP is infeasible if $\mathcal{C} = \emptyset$ or $Trace(W) - \sum_{i=1}^{N} x_i^2 > 0$ for any point $(\mathbf{x}, W)$ in $\mathcal{C} \neq \emptyset$.*

Proof If $\mathcal{C} = \emptyset$, then the system (6) is obviously infeasible. Otherwise, suppose $(\mathbf{x}, W) \in \mathcal{C} \neq \emptyset$, applying Schur complements to $\begin{bmatrix} W & \mathbf{x} \\ \mathbf{x}^T & 1 \end{bmatrix} \succeq 0$, we have $W - \mathbf{x}\mathbf{x}^T \succeq 0$ which means $Trace(W - \mathbf{x}\mathbf{x}^T) = Trace(W) - \sum_{i=1}^{N} x_i^2 \geq 0$. This inequality is held for every point in $\mathcal{C}$. Combine this inequality with the nonconvex set $\mathcal{RC}$, we must have $Trace(W) - \sum_{i=1}^{N} x_i^2 = 0$ for any feasible solution of the system (6). Clearly, if $Trace(W) - \sum_{i=1}^{N} x_i^2 > 0$ for all points $(\mathbf{x}, W)$ in $\mathcal{C}$, the system (6) must be infeasible (i.e. an empty set). □

Now, we can give a concave minimization problem as

$$\begin{array}{ll} \min & f(\mathbf{x}, W) = Trace(W) - \sum_{i=1}^{N} x_i^2 \\ s.t. & Q_L(\mathbf{x}, W) := Q_0 + \sum_{i=1}^{N} x_i Q_i + \sum_{i,j=1}^{N} w_{ij} Q_{ij} \preceq 0 \\ & \begin{bmatrix} W & \mathbf{x} \\ \mathbf{x}^T & 1 \end{bmatrix} \succeq 0 \\ & (\mathbf{x}, W) \in \mathcal{H} \times S_+^N. \end{array} \quad (7)$$

$$\mathcal{C} := \{(\mathbf{x}, W) \in \mathcal{H} \times S_+^N : Q_L(\mathbf{x}, W) := Q_0 + \sum_{i=1}^{N} x_i Q_i + \sum_{i,j=1}^{N} w_{ij} Q_{ij} \preceq 0; \begin{bmatrix} W & \mathbf{x} \\ \mathbf{x}^T & 1 \end{bmatrix} \succeq 0\}$$

From lemma 1, we know that the formulation (3) is feasible if and only if 0 is the optimal objective value of the concave minimization problem (7).

Generally speaking, the BMIFP (1) is feasible if and only if the QMIFP (2) is feasible, the QMIFP (2) is equivalent to the formulation (3), and the formulation (3) is feasible if and only if 0 is the optimal objective value of the concave minimization problem (7). In one word, we can solve the concave programming (7) to determine whether the BMIFP (1) is feasible or not. It is easy to verify that if $(\mathbf{x}, W)$ is feasible

to (7) such that $f(\mathbf{x}, W) \leq 0$, then $\mathbf{x}$ must be feasible to the original BMIFP (1).

Observing the problem (7), the objective function $f(\mathbf{x}, W)$ is a quadratic concave function since $-\sum_{i=1}^{N} x_i^2$ is quadratic concave and $Trace(W)$ is linear. The constraint set of (7) is convex since

- LMI constraint gives convex set.
- The hyper-rectangle $\mathcal{H}$ is also a convex set.
- S_+^N is a convex cone.

The intersection of these convex sets is also a convex set. Moreover, all variables $(\mathbf{x}, W)$ of the problem (6) are bounded. Because $\mathbf{x} \in \mathcal{H}$ is bounded, and all w_{ij}'s are also bounded since $w_{ij} := x_i x_j$. Therefore, we can introduce the upper bound and lower bound of the variable w_{ij} as

$$\underline{w}_{ij} \leq w_{ij} \leq \bar{w}_{ij} (i, j = 1, \ldots, N) \tag{8}$$

where

$$\underline{w}_{ij} = \min\{\underline{x}_i \underline{x}_j, \underline{x}_i \bar{x}_j, \bar{x}_i \underline{x}_j, \bar{x}_i \bar{x}_j\},$$
$$\bar{w}_{ij} = \max\{\underline{x}_i \underline{x}_j, \underline{x}_i \bar{x}_j, \bar{x}_i \underline{x}_j, \bar{x}_i \bar{x}_j\}.$$

Let $\mathcal{H}_W := \{(\mathbf{x}, W) \in \mathbb{R}^N \times S_+^N : \underline{\mathbf{x}} \leq \mathbf{x} \leq \bar{\mathbf{x}}, \underline{w}_{ij} \leq w_{ij} \leq \bar{w}_{ij}, i, j = 1, \ldots, N\}$. Clearly, all feasible solution $(\mathbf{x}, W)$ of (7) must be inside of $\mathcal{H}_W$ although $\mathcal{H}_W \subset \mathcal{H} \times S_+^N$. Moreover, $\mathcal{H}_W$ is a compact convex set.

On the other hand, there are two LMIs in the problem (7). We can reformulate several LMIs into one LMI according to the lemma 2.

Lemma 2 *Suppose that we have p LMIs as*

$$F^j(\mathbf{x}) := F_0^j + \sum_{i=1}^{N} x_i F_i^j \preceq 0, (j = 1, \ldots, p). \tag{9}$$

We can reformulate them as just one equivalent LMI

$$\hat{F}(\mathbf{x}) := \hat{F}_0 + \sum_{i=1}^{N} x_i \hat{F}_i \preceq 0 \tag{10}$$

where

$$\hat{F}_i = \begin{bmatrix} F_i^1 & & 0 \\ & \ddots & \\ 0 & & F_i^p \end{bmatrix}, (i = 0, \ldots, N).$$

Proof Clearly, the formula (10) is exactly the formula (11)

$$\hat{F}(\mathbf{x}) := \begin{bmatrix} F^1(\mathbf{x}) & & 0 \\ & \ddots & \\ 0 & & F^p(\mathbf{x}) \end{bmatrix} \preceq 0. \tag{11}$$

If $\mathbf{x}^*$ is feasible to (9), i.e. $F^j(\mathbf{x}^*) \preceq 0, (j = 1, \ldots, p)$, obviously, it is also feasible to the formula (11). Inversely, if $\mathbf{x}^*$ is feasible to (11), it is well-known that all principal minors of $\hat{F}(\mathbf{x}^*)$ have to be negative semidefinite, i.e. $F^j(\mathbf{x}^*) \preceq 0, (j = 1, \ldots, p)$. □

Therefore, because W is a symmetric matrix, we represent the LMI $\begin{bmatrix} W & \mathbf{x} \\ \mathbf{x}^T & 1 \end{bmatrix} \succeq 0$ as the standard form:

$$F(\mathbf{x}, W) := F_0 + \sum_{i=1}^{N} x_i F_i + \sum_{i,j=1}^{N} w_{ij} F_{ij} \preceq 0 \tag{12}$$

where

$$F_0 := \begin{bmatrix} 0 & 0 \\ 0 & -1 \end{bmatrix}; F_i := \begin{bmatrix} 0 & -e_i \\ -e_i^T & 0 \end{bmatrix} (i = 1, \ldots, N); F_{ij} := \begin{bmatrix} M_{ij} & 0 \\ 0 & 0 \end{bmatrix} (i, j = 1, \ldots, N).$$

The matrix M_{ij} is $N \times N$ symmetric matrix whose (i, j) and (j, i) elements, if $i \neq j$, are $-\frac{1}{2}$, zero otherwise. If $i = j$, M_{ii} becomes a diagonal matrix whose i-th element in the diagonal is -1, zero otherwise, i.e. $M_{ii} := diag(e_i)$, where $e_i \in \mathbb{R}^N$ is the i-th canonical base vector of the space $\mathbb{R}^N$. Note that, the matrices F_i's and F_{ij}'s are very sparses (at most 2 nonzero elements for each one). So we can construct and handle them efficiently with sparse matrix form in computation.

In virtue of Lemma (2), we can represent the all LMI constraints of (7) into one LMI as

$$\hat{Q}(\mathbf{x}, W) := \hat{Q}_0 + \sum_{i=1}^{N} x_i \hat{Q}_i + \sum_{i,j=1}^{N} w_{ij} \hat{Q}_{ij} \preceq 0 \tag{13}$$

where

$$\hat{Q}_0 := \begin{bmatrix} Q_i & 0 \\ 0 & F_i \end{bmatrix}, (i = 0, \ldots, N); \hat{Q}_{ij} := \begin{bmatrix} Q_{ij} & 0 \\ 0 & Fij \end{bmatrix} (i, j = 1, \ldots, N).$$

Note that, Q_i and Q_{ij} belong to the symmetric matrix space S^k, while F_i and F_{ij} belong to S^{N+1}. Hence, $\hat{Q}_i$ and $\hat{Q}_{ij}$ belong to S^{k+N+1}, and they are all sparse matrices.

Finally, we can represent the problem (7) as the following equivalent simplified form:

$$\begin{array}{ll} \min & f(\mathbf{x}, W) = Trace(W) - \sum_{i=1}^{N} x_i^2 \\ s.t. & \hat{Q}(\mathbf{x}, W) := \hat{Q}_0 + \sum_{i=1}^{N} x_i \hat{Q}_i + \sum_{i,j=1}^{N} w_{ij} \hat{Q}_{ij} \preceq 0 \\ & (\mathbf{x}, W) \in \mathcal{H}_W. \end{array} \tag{14}$$

In the problem (14), we minimize a quadratic concave function with a compact convex set. Clearly, it belongs to the realm of DC optimization.

We have the equivalent *unconstrained DC programming representation* of the problem (14) as

$$\min \quad f(\mathbf{x}, W) = g(\mathbf{x}, W) - h(\mathbf{x}, W). \tag{15}$$

where $g(\mathbf{x}, W) := \chi_{\mathcal{C}}(\mathbf{x}, W)$ is a convex function ($\chi_{\mathcal{C}}(\mathbf{x}, W)$ denote the indicator function defines as $\chi_{\mathcal{C}}(\mathbf{x}, W) := 0$ if $(\mathbf{x}, W) \in \mathcal{C}$, $+\infty$ otherwise). $\mathcal{C} := \{(\mathbf{x}, W) \in \mathcal{H}_W, \hat{Q}(\mathbf{x}, W) := \hat{Q}_0 + \sum_{i=1}^{N} x_i \hat{Q}_i + \sum_{i,j=1}^{N} w_{ij} \hat{Q}_{ij} \preceq 0\}$ is a compact convex set.
$h(\mathbf{x}, W) := \sum_{i=1}^{N} (x_i^2 - w_{ii})$ is a quadratic convex function.

4 DC Programming and DCA

In order to give readers a brief view of DC programming and DCA, we will outline some essential theoretical and algorithmic results in this section. There are also many mathematical definitions frequently used in convex analysis, the reader is referred to the reader is referred to [32,33]. For more details about DC programming and DCA, the reader is referred to ([17–28], [34–36]) and the references therein.

4.1 General D.C. programming

Let $X = \mathbb{R}^n$. Y is the dual space of X, Y can be identified with X. $\Gamma_0(X)$ denote the cone of lower semi-continuous and proper convex functions on X. $DC(X) := \Gamma_0(X) - \Gamma_0(X)$ denote the vector space of DC functions defined on X which is quite large to contain most of objective functions of real life optimization problems. The conjugate function ϕ^* of $\phi \in \Gamma_0(X)$ belongs to $\Gamma_0(Y)$ and defined by

$$\phi^*(y) := \sup\{< x, y > -\phi(x) : x \in X\}.$$

The DC optimization problem is given as

$$(P_{dc}) \qquad \alpha = \inf\{f(x) = g(x) - h(x) : x \in X\}$$

where $g, h \in \Gamma_0(X)$. We admit a nature convention in DC programming that is $(+\infty) - (+\infty) = (+\infty)$.
The dual problem of P_{dc} is also a DC program with the same optimal value defined by:

$$(D_{dc}) \qquad \alpha = \inf\{h^*(y) - g^*(y) : y \in Y\}.$$

We observe the perfect symmetry between primal and dual DC programs: the dual of (D_{dc}) is exactly (P_{dc}).

Let C be a closed convex subset of X and $k, h \in \Gamma_0(X)$. The constrained DC program

$$\inf\{f(x) = k(x) - h(x) : x \in C\}$$

can be reformulated as (P_{dc}) with an additional indicator function $\chi_C(x)$ incorporated in the first component k as $g(x) := k(x) + \chi_C(x)$, where $\chi_C(x) := 0$ if $x \in C$, $+\infty$ otherwise.

A DC program is called *polyhedral* if either g or h is a polyhedral convex function. This class of DC programs is frequently encountered in realistic optimization problems and has been extensively developed in our previous works (see e.g. [22] and references therein), enjoys interesting properties (from both theoretical and practical viewpoints) concerning the local optimality and the convergence of DCA.

Recall that $\forall \theta \in \Gamma_0(\mathbb{R}^n)$ and $x_0 \in dom(\theta) := \{x \in \mathbb{R}^n : \theta < +\infty\}$, $\partial\theta(x_0)$ denotes the subdifferential of θ at x_0.

$$\partial\theta(x_0) := \{y \in \mathbb{R}^n : \theta(x) \geq \theta(x_0) + < x - x_0, y >, \forall x \in \mathbb{R}^n\}.$$

Note that $\partial\theta(x_0)$ is a closed convex subset of $\mathbb{R}^n$. If θ is differentiable at x_0, then $\partial\theta(x_0)$ reduces to a singleton which is exactly $\{\nabla\theta(x_0)\}$.

Let $\mathcal{P}$ and $\mathcal{D}$ be the solution sets of (P_{dc}) and (D_{dc}) respectively. Then $\partial h(x) \subset \partial g(x), \forall x \in \mathcal{P}$ and $\partial g^*(y) \subset \partial h^*(y), \forall y \in \mathcal{D}$. They are necessary local optimality conditions for (P_{dc}) and (D_{dc}).

A point x^* verifies the condition $\partial h(x^*) \cap \partial g(x^*) \neq \emptyset$ is called a critical point of $g - h$. The necessary local optimality conditions mentioned above are also sufficient for many classes of DC programs quite often encountered in practice, such as the case when the function f being locally convex at x^*, and the case when the problem is a polyhedral DC program with h being a polyhedral convex function.

The transportation of global solutions between (P_{dc}) and (D_{dc}) can be expressed by:

$$[\bigcup_{y^* \in \mathcal{D}} \partial g^*(y^*)] \subset \mathcal{P}, [\bigcup_{x^* \in \mathcal{P}} \partial h(x^*)] \subset \mathcal{D}.$$

Under technical conditions, this transportation holds also for local solutions of (P_{dc}) and (D_{dc}) (see [17–19, 24]).

4.2 DC Algorithm (DCA)

DCA was first introduced by Pham Dinh Tao in 1985 as an extension of the subgradient algorithm, and extensively developed by Le Thi Hoai An and Pham Dinh Tao since 1993 to solve DC programs. It is actually one of the rare algorithms for nonlinear nonconvex nonsmooth programming which allows solving very efficiently large-scale DC programs. DCA has successfully been applied in solving real world nonconvex programs to which it quite often gives global solutions and proved to be more robust and more efficient than related standard methods, especially for large-scale problems.

Based on local optimality conditions and duality in DC programming, the DCA consists in the construction of two sequences $\{x^k\}$ and $\{y^k\}$ (candidates to be solutions of (P_{dc}) and (D_{dc}) resp.) such that the sequences $\{g - h(x^k)\}$ and $\{h^* - g^*(y^k)\}$ are decreasing, and x^{k+1} (resp. y^k) is an optimal solution of the convex program (P_k) (resp. (D_k)) defined by:

$$\inf\{g(x) - [h(x^k) + < x - x^k, y^k >] : x \in \mathbb{R}^n\} \qquad (P_k)$$

$$\inf\{h^*(y) - [g^*(y^{k-1}) + < y - y^{k-1}, x^k >] : y \in \mathbb{R}^n\} \qquad (D_k)$$

where (P_k) (resp. (D_k)) is derived from (P_{dc}) (resp. (D_{dc})) by replacing the component h (resp. g^*) with its affine minorization at a neighborhood of x^k (resp. y^{k-1}), which is exactly the first order Taylor series expansion of h (resp. g^*) in a neighborhood of x^k (resp. y^{k-1}).

We have the general convergence result that x^k (resp. y^k) converges to a primal feasible solution $\tilde{x}$ (resp. a dual feasible solution $\tilde{y}$), verifying local optimality conditions. $\tilde{x}$ (resp. $\tilde{y}$) is also a critical point of $g - h$ (resp. $h^* - g^*$).

The DCA yields the next scheme:

$$y^k \in \partial h(x^k); \quad x^{k+1} \in \partial g^*(y^k).$$

Note that, the general framework of DCA can only applicable to *DC programs within convex set*. If the constraint set is nonconvex, DCA can not be applied directly, some classes of nonconvex constrained DC problems have already been investigated in our many previous works (see e.g. [23], [28] etc.).

We are going to end this section by summarizing the DCA's Convergence Theorem.

Theorem 2 (Convergence Theorem of DCA) *(See [17–19]) DCA is a descent method without linesearch which enjoys the following primal properties (the dual ones can be formulated in a similar way):*

1. *The sequences* $\{g(x^k) - h(x^k)\}$ *and* $\{h^*(y^k) - g^*(y^k)\}$ *are decreasing and*
   - $g(x^{k+1}) - h(x^{k+1}) = g(x^k) - h(x^k)$ *if and only if* $y^k \in \partial g(x^k) \cap \partial h(x^k)$, $y^k \in \partial g(x^{k+1}) \cap \partial h(x^{k+1})$ *and* $[\rho(g, C) + \rho(h, C)]\|x^{k+1} - x^k\| = 0$. *Moreover if* g *or* h *are strictly convex on* C, *then* $x^k = x^{k+1}$. *In such a case DCA terminates at finitely many iterations.*
     Here C *(resp.* D*) denotes a convex set containing the sequence* $\{x^k\}$ *(resp.* $\{y^k\}$*) and* $\rho(g, C)$ *denote the modulus of strong convexity of* g *on* C *given by:*
     $$\rho(g, C) := \sup\{\rho \geq 0 : g - (\rho/2)\|.\|^2 \text{ be convex on } C\}.$$
   - $h^*(y^{k+1}) - g^*(y^{k+1}) = h^*(y^k) - g^*(y^k)$ *if and only if* $x^{k+1} \in \partial g^*(y^k) \cap \partial h^*(y^k)$, $x^{k+1} \in \partial g^*(y^{k+1}) \cap \partial h^*(y^{k+1})$ *and* $[\rho(g^*, D) + \rho(h^*, D)]\|y^{k+1} - y^k\| = 0$. *Moreover if* g^* *or* h^* *are strictly convex on* D, *then* $y^k = y^{k+1}$. *In such a case DCA terminates at finitely many iterations.*

2. *If* $\rho(g, C) + \rho(h, C) > 0$ *(resp.* $\rho(g^*, D) + \rho(h^*, D) > 0$*) then the sequence* $\{\|x^{k+1} - x^k\|^2\}$ *(resp.* $\{\|y^{k+1} - y^k\|^2\}$*) converges.*
3. *If the optimal value of* P_{dc} *is finite and the infinite sequence* $\{x^k\}$ *and* $\{y^k\}$ *are bounded then every limit point* x^∞ *(resp.* y^∞*) of the sequence* $\{x^k\}$ *(resp.* $\{y^k\}$*) is a critical point of* $g - h$ *(resp.* $h^* - g^*$*).*
4. *DCA is a descent method without linesearch which has a linear convergence for general DC programs.*
5. *DCA has a finite convergence for polyhedral DC programs.*

Remark 1 Every DC function has infinitely many DC decompositions which have crucial effects on the DCA's convergence speed, efficiency, robustness, globality of computed solution.

Remark 2 In practice, DCA can tackle some classes of large-scale nonconvex, nonsmooth program and converges often to global solutions.

Remark 3 The choice of a good initial point is still an open question for DCA, it depends on the specific structure and features of the DC program.

Remark 4 Quality of computed solutions by DCA: the lower the corresponding value of the objective is, the better the local algorithm will be.

Remark 5 The degree of dependence on initial points: the larger the set (made up of starting points which ensure convergence of the algorithm to a global solution) is, the better the algorithm will be.

4.3 Geometric interpretation for DCA

In the algorithm of DCA, at the k-th iteration, we will construct an upper bound convex function $g(x) - (h(x^k) + < x - x^k, y^k >)$, with the iteration point x^k and $y^k \in \partial g(x^k)$, for the original DC function $g(x) - h(x)$. It is easy to verify the following proposition:

Proposition 3 *Suppose g is differentiable. Let $f^k(x) := g(x) - [h(x^k) + < x - x^k, y^k >]$ and $f(x) = g(x) - h(x)$. We have*

- $f^k(x) \geq f(x), \forall : x \in \mathcal{C}$.
- $f^k(x^k) = f(x^k)$.
- $\nabla f^k(x^k) = \nabla f(x^k)$.

Proof Since $h(x)$ is a convex function in $\mathcal{C}$, it is well known that $h(x) \geq h(x^k) + \langle x - x^k, y^k \rangle, \forall x \in \mathcal{C}$. Therefore we have $g(x) - [h(x^k) + \langle x - x^k, y^k \rangle] \geq g(x) - h(x), \forall x \in \mathcal{C}$, i.e. $f^k(x) \geq f(x), \forall : x \in \mathcal{C}$.
$f^k(x^k) = g(x^k) - [h(x^k) + \langle x^k - x^k, y^k \rangle] = g(x^k) - h(x^k) = f(x^k)$.
$\nabla f^k(x) = \nabla g(x) - y^k$. Because g is differentiable, we have $y^k = \nabla h(x^k)$. Hence, we have $\nabla f^k(x) = \nabla g(x) - \nabla h(x^k)$. Then $\nabla f^k(x^k) = \nabla g(x^k) - \nabla h(x^k) = \nabla f(x^k)$. □

Proposition 3 gives us a geometric view to the relationship between f^k and f. We can see that f^k is coinciding with f at the point x^k, and f^k is located above the function f at all point of $\mathcal{C}$. A roughly speaking is that the surface of the function f^k could be imagined as a "bowl" (opens upward) locating on the top of the surface of f and touch with f at the point $(x^k, f(x^k))$. The principle of the algorithm DCA can be explained in the following illustrated picture:

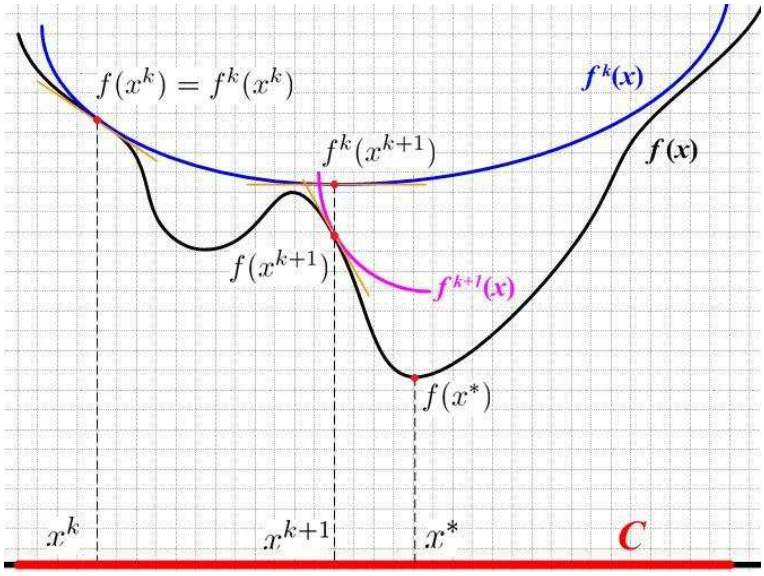


Fig. 1 Geometric interpretation of DCA

We can observe that how DCA to find the iteration point x^{k+1} from x^k and so on. Some important features of DCA have also an easy explanation via this figure. For instance, we know that DCA will generate a decreasing sequence $\{f(x^k)\}$. This can be observed directly in the figure. Since x^{k+1} is a minimum of the convex function f^k over the convex set $\mathcal{C}$. Therefore, $f^k(x^{k+1}) \leq f^k(x^k) = f(x^k)$. Moreover, in virtue of the proposition 3, we have $f^k(x) \geq f(x), \forall x \in \mathcal{C}$, i.e. $f^k(x^{k+1}) \geq f(x^{k+1})$. Finally, we have $f(x^{k+1}) \leq f^k(x^{k+1}) \leq f(x^k)$. This means that $\{f(x^k)\}$ is a decreasing sequence.

We can also observe in the figure that DCA is possible to jump out a neighborhood of a local optimal solution. For instance, DCA jumps over a local minimum between x^k and x^{k+1}, and goes to a neighborhood of the global optimal solution x^*. This special feature is very important and interesting for global optimization, although it can not be always guaranteed during iterations of DCA. However, we can understand that this phenomenon depends on the location of the initial point and the nature of the upper bound convex function f^k. That is to say that the performance of DCA depends on the choice of an initial point and the structure of the DC decomposition. These two questions must be crucially considered when applying DCA. Furthermore, the convergence of DCA can be also pointed out in the figure since $\{f(x^k)\}$ is a decreasing sequence and bounded below, i.e. the sequence $\{f(x^k)\}$ is convergent.

5 DCA for solving QMIFP

In this section, we investigate how to apply DCA for solving QMIFP (2). We have reformulated (2) as a DC program (15). For convenience discourse, we rewrite the DC program as:

$$\min \quad f(\mathbf{x}, W) = g(\mathbf{x}, W) - h(\mathbf{x}, W). \tag{16}$$

where $h(\mathbf{x}, W) := \sum_{i=1}^{N}(x_i^2 - w_{ii})$ is a continuous and differentiable quadratic convex function. $g(\mathbf{x}, W) := \chi_{\mathcal{C}}(\mathbf{x}, W)$ is a convex function.
$\mathcal{C} := \{(\mathbf{x}, W) \in \mathcal{H}_W, \hat{Q}(\mathbf{x}, W) := \hat{Q}_0 + \sum_{i=1}^{N} x_i\hat{Q}_i + \sum_{i,j=1}^{N} w_{ij}\hat{Q}_{ij} \preceq 0\}$ is a compact convex set (here $\mathcal{H}_W := \{(\mathbf{x}, W) \in \mathbb{R}^N \times S_+^N : \underline{\mathbf{x}} \leq \mathbf{x} \leq \bar{\mathbf{x}}, \underline{w}_{ij} \leq w_{ij} \leq \bar{w}_{ij}, i, j = 1, \ldots, N\}$).

According to the general framework of DCA, we should construct two sequences $\{(\mathbf{x^k}, W^k)\}$ and $\{(\mathbf{y^k}, Z^k)\}$ via the following scheme:

$$\begin{array}{c} (\mathbf{x}^k, W^k) \longrightarrow (\mathbf{y}^k, Z^k) \in \partial h(\mathbf{x}^k, W^k) \\ \qquad\qquad \swarrow \\ (\mathbf{x}^{k+1}, W^{k+1}) \in \partial g^*(\mathbf{y}^k, Z^k). \end{array} \tag{17}$$

Since $h(\mathbf{x}, W) := \sum_{i=1}^{N}(x_i^2 - w_{ii})$ is a continuous and differentiable quadratic convex function, then $\partial h(\mathbf{x}^k, W^k)$ reduces to a singleton $\{\nabla h(\mathbf{x}^k, W^k)\}$ which could be computed explicitly as:

$$\begin{array}{rl} (\mathbf{y}^k, Z^k) \in \partial h(\mathbf{x}^k, W^k) & \Leftrightarrow \{(\mathbf{y}^k, Z^k) = \nabla h(\mathbf{x}^k, W^k)\} \\ & \Leftrightarrow \{\mathbf{y^k} = 2\mathbf{x^k}; Z^k = -I_N\}. \end{array} \tag{18}$$

where I_N is the unit matrix of dimension $N \times N$.

For computing $(\mathbf{x}^{k+1}, W^{k+1}) \in \partial g^*(\mathbf{y}^k, Z^k)$, we can solve the following program:

$$(\mathbf{x}^{k+1}, W^{k+1}) \in argmin\{Trace(W) - \langle \mathbf{y^k}, \mathbf{x} \rangle : (\mathbf{x}, W) \in \mathcal{C}\}. \tag{19}$$

Finally, DCA yields the following iterative scheme:

$$(\mathbf{x}^{k+1}, W^{k+1}) \in argmin\{Trace(W) - \langle 2\mathbf{x^k}, \mathbf{x} \rangle : (\mathbf{x}, W) \in \mathcal{C}\}. \tag{20}$$

The program (20) is to minimize a linear function over the convex set $\mathcal{C}$. It is a *linear LMI program.* It is well known that $LMI \equiv SDP$ (SemiDefinite Programming), and many efficient SDP solvers such as SDPA, SeDuMi, SDPT, Yalmip could be used.

For stopping criterion, the objective function $f(\mathbf{x}, W)$ must be non-negative over the constraint of (7) since $\begin{bmatrix} W & \mathbf{x} \\ \mathbf{x}^T & 1 \end{bmatrix} \succeq 0 \Rightarrow W - \mathbf{x}\mathbf{x}^T \succeq 0 \Rightarrow f(\mathbf{x}, W) := Trace(W) - \sum_{i=1}^{N} x_i^2 = Trace(W - \mathbf{x}\mathbf{x}^T) \geq 0$. Therefore, a BMIFP or QMIFP is feasible if and only if there exists a feasible solution $(\tilde{\mathbf{x}}, \tilde{W}) \in \mathcal{C}$ such that $f(\tilde{\mathbf{x}}, \tilde{W}) = 0$.
Then we can describe the first stopping criterion as:
Stopping Criterion 1: *DCA will stop at k-th iteration if $f(\mathbf{x}^k, W^k) \leq \epsilon_1$ (where ϵ_1 is a given small positive number, such as 10^{-6}).*

On the other hand, $f(\mathbf{x}^k, W^k)$ is always far from zero during the iterations of DCA (For instance, BMIFP is infeasible). In this case, we propose the second stopping criterion as:
Stopping Criterion 2: *DCA can stop when $\frac{|f(\mathbf{x}^k, W^k) - f(\mathbf{x}^{k-1}, W^{k-1})|}{|f(\mathbf{x}^k, W^k)|} \leq \epsilon_2$ (where ϵ_2 is a given small positive number, such as 10^{-5}).*

Now, we can describe our DCA for solving the QMIFP as:

DCA1

Initialization:
Choose an initial point $\underline{\mathbf{x}} \leq \mathbf{x^0} \leq \bar{\mathbf{x}}$, compute $W^0 = \mathbf{x}^0\mathbf{x}^{0T}$.
Let ϵ_1, ϵ_2 be sufficiently small positive numbers.
Let iteration number $k = 0$.

Iteration: $k = 0, 1, 2, ...$
Solve the SDP program (20) for computing $(\mathbf{x}^{k+1}, W^{k+1})$ from $(\mathbf{x}^k, W^k)$.

stopping criterions:
If $f(\mathbf{x}^{k+1}, W^{k+1}) \leq \epsilon_1$ then
  Stop iteration. QMIFP is feasible and $(\mathbf{x}^{k+1}, W^{k+1})$ is a feasible solution.
Else if $|f(\mathbf{x}^{k+1}, W^{k+1}) - f(\mathbf{x}^k, W^k)| \leq \epsilon_2 |f(\mathbf{x}^k, W^k)|$ then
  Stop iteration. No feasible solution is found.
Otherwise, let $k = k + 1$ and repeat `Iteration`.

Theorem 3 (Convergence Theorem of DCA1) *DCA1 can generate a sequence* $\{(\mathbf{x}^k, W^k)\}$ *such that the sequence* $\{f(\mathbf{x}^k, W^k)\}$ *is decreasing and bounded below (i.e. convergent).*

Proof The proof of this theorem is based on the proof of the general convergence theorem of DCA which could be found in the papers [17–19]. Since DCA1 will become to a general DCA without the Stopping Criterion 1. It is clear that this additional stopping criterion will not change the convergence property of DCA, it will only change the quality of the computed solution, i.e. we don't have the result that DCA1 could converge to a KKT point.

6 For solving a large-scale DC program

In the algorithm DCA1, we need to solve one SDP subproblem at each iteration. The computational time will be expensive when this subproblem is large dimensional. We have tested several SDP solvers, SDPA, SeDuMi, CSDP and DSDP. To our knowledge, they are the most popular and high-powered SDP solvers at the present time. In general, solving a SDP of 500 variables and 300 linear constraints will take about $15 - 20$ seconds. The SDP subproblem of DCA has $m + n + \frac{(m+n)(m+n+1)}{2}$ variables and one LMI constraint of order $[3(m+n)+k+1] \times [3(m+n)+k+1]$. When m, n and k take large value, solving such a large-scale SDP problem will be very expensive in computational time. In order to reduce the computational time for each iteration, we propose to solve the SDP subproblem *partially* instead of entirely. More specifically, we can control the SDP solver's computing procedure and stop the computation when some conditions satisfied. In fact, in the DCA procedure, the objective for solving a SDP subproblem is to find a better feasible solution $(\mathbf{x}^{k+1}, W^{k+1})$ from the current feasible solution $(\mathbf{x}^k, W^k)$, i.e. the value of the DC function $g-h$ reduces. Therefore, we don't need to find a global optimal solution of the SDP subproblem at each iteration, it is sufficient to find a better feasible solution. Therefore, we can stop the SDP solution procedure when a better feasible solution is found. The proposed strategy will be helpful for solving large-scale problem, since the computational time for each iteration of DCA will be less expensive.

Here is the modified DCA:

DCA2

Initialization:
Choose an initial point $\underline{\mathbf{x}} \leq \mathbf{x^0} \leq \bar{\mathbf{x}}$, compute $W^0 = \mathbf{x}^0\mathbf{x}^{0T}$.
Let ϵ_1, ϵ_2 be sufficiently small positive numbers.
Let iteration number $k = 0$.

Iteration: $k = 0, 1, 2, ...$
Applying one SDP algorithm (such as Interior point method)
for solving (20) step by step until we found a better point
whose objective value $f(\mathbf{x}^{k+1}, W^{k+1})$ is smaller than $f(\mathbf{x}^k, W^k)$.

stopping criterions:
If $f(\mathbf{x}^{k+1}, W^{k+1}) \leq \epsilon_1$ then
  Stop iteration. QMIFP is feasible and $(\mathbf{x}^{k+1}, W^{k+1})$ is a feasible solution.
Else if $|f(\mathbf{x}^{k+1}, W^{k+1}) - f(\mathbf{x}^k, W^k)| \leq \epsilon_2 |f(\mathbf{x}^k, W^k)|$ then
  Stop iteration. No feasible solution is found.
Otherwise, let $k = k + 1$ and repeat `Iteration`.

Theorem 4 (Convergence Theorem of DCA2) *DCA2 can generate a sequence* $\{(\mathbf{x}^k, W^k)\}$ *such that the sequence* $\{f(\mathbf{x}^k, W^k)\}$ *is decreasing and bounded below (i.e. convergent).*

Proof The proof of this theorem is quite similar to the proof the convergence theorem of DCA1. The only different between these two methods is that DCA2 does not require globally solving a SDP problem at each iteration. In fact, globally solving a convex SDP subproblem is not necessary. If we can find a better feasible solution at each iteration, the convergence property of the DCA will not be changed.

7 Hybrid Algorithm DCA-B&B for solving QMIFP

We must emphasize that, from the theoretical viewpoint, if DCA is stopped by satisfying the second stopping criterion. It is not a sufficient to decide that the problem QMIFP is infeasibility or not, since DCA is only *a local optimization method*. We must recur to some global optimization techniques for further research. In this section, we will propose a hybrid method of combining DCA with a Branch-and-Bound (B&B) scheme.

The Branch-and-Bound (B&B) method used in this paper is an alterative procedure in which the hyper-rectangle of x-space, $\mathcal{H} := \{\mathbf{x} \in \mathbb{R}^N : \underline{\mathbf{x}} \leq \mathbf{x} \leq \bar{\mathbf{x}}\}$, will be iteratively partitioned into small sets (*branching procedure*), and the search over each partition set M is carried out though estimating a lower bound $\beta(M)$ of $f(\mathbf{x}, W)$ over $\mathcal{C} \cap M$, then computing a upper bound $\alpha(M)$ via DCA (*bounding procedure*).

Clearly, those partition sets M with $\beta(M) > 0$ cannot provide any feasible solution. These partitions should be discarded from further consideration. On the other hand, the partition set with smallest $\beta(M)$ can be considered as the most promising one which should be subdivided in the next step into more refined subsets for further investigation. For those partition sets M with $\beta(M) \leq 0$, local search via DCA will be carried out in order to reduce the upper bound on these partition sets. For any partition M in which DCA finds a upper bound $\alpha(M) = 0$, the B&B could be terminated and QMIFP is feasible. Oppositely, if all partitions sets are discarded and no feasible solution has been found by DCA, then QMIFP is infeasible.

Given a partition set $M := [p, q] = \{\mathbf{x} \in \mathbb{R}^N : p \leq \mathbf{x} \leq q\} \subseteq \mathcal{H}$. The optimization problem on M should be defined as:

$$\begin{array}{ll} \min & f(\mathbf{x}, W) := Trace(W) - \sum_{i=1}^N x_i^2 \\ s.t. & (\mathbf{x}, W) \in \mathcal{C} \cap M. \end{array} \tag{21}$$

The convex underestimation of f on the partition set $M := [p, q]$ is defined as $F_M(\mathbf{x}, W) := Trace(W) + \langle \mathbf{c}, \mathbf{x} \rangle + d$ where $\mathbf{c} := -(p + q)$, $d := \langle p, q \rangle$. In fact, F_M

is exactly the convex envelope of f defined on M. Therefore, we can define the lower bound estimation problem on the partition M as:

$$\beta(M) = \min\{F_M(\mathbf{x}, W) := Trace(W) + \langle \mathbf{c}, \mathbf{x}\rangle + d : (\mathbf{x}, W) \in \mathcal{C} \cap M\}. \tag{22}$$

On the other hand, DCA applies to (21) yields the following iterative scheme:

$$(\mathbf{x}^{k+1}, W^{k+1}) \in argmin\{Trace(W) - \langle 2\mathbf{x}^k, \mathbf{x}\rangle : (\mathbf{x}, W) \in \mathcal{C} \cap M\}. \tag{23}$$

These iteration points $(\mathbf{x}^{k+1}, W^{k+1})$ provides upper bounds $\alpha(M) := f(\mathbf{x}^{k+1}, W^{k+1})$, and the sequence $\{f(\mathbf{x}^{k+1}, W^{k+1})\}$ is decreasing and convergent.

Our DCA-B&B method for solving QMIFP (2) involves three basic operations:

1. **Branching**: At each iteration of B&B, a partition set $M = [p, q]$ is selected and subdivided into two subsets via some hyperplane defined as $x_{i_M} = \lambda$ where i_M and $\alpha \in (p_{i_M}, q_{i_M})$ are chosen by some specified rules. It is well known that, when $i_M = argmax_{\{j=1,\ldots,N\}}\{q_j - p_j\}$ and $\lambda = (p_{i_M} + q_{i_M})/2$ the subdivision is called the *standard bisection*. In our paper, instead of using standard bisection, we would like to use *an adaptive subdivision rule* with consideration the information so far obtained during the exploration on the partition M. The reason is due to the fact that the convergence of a B&B scheme can be significantly faster by using an adaptive subdivision rule than using the standard bisection rule.

   Adaptive subdivision rule: Given a partition $M = [p, q]$. Using DCA applied to (21) to obtain a computed solution $(\mathbf{x}^*, W^*)$. Suppose it is not a feasible solution to QMIFP. We will compute i_M and λ as follows:
   - $(i^*, j^*) \in argmax\{|w^*_{ij} - x^*_i x^*_j| : \forall i, j = 1, \ldots, N\}$.
   - $i_M = argmax\{|p_{i^*} - q_{i^*}|, |p_{j^*} - q_{j^*}|\}$.
   - $\lambda = (p_{i_M} + q_{i_M})/2$.

   Then, the partition set M will be divided into two subsets by the line $x_{i_M} = \lambda$. The objective of this adaptive subdivision rule is to reduce the gaps between w_{ij} and $x_i x_j$ for all $i, j = 1, \ldots, N$, since a feasible solution $(\mathbf{x}, W)$ of QMIFP must satisfy the equality $W = xx^T$.

2. **Bounding**: On a partition M, The lower bound $\beta(M)$ should be computed by solving the SDP lower bound estimation problem (22). The upper bound $\alpha(M)$ on a partition M will be computed by DCA via (23) to obtain a computed solution $(\mathbf{x}^*, W^*)$, then $\alpha(M) := f(\mathbf{x}^*, W^*)$.

3. **Discarding**: A partition M will be discarded if one of the conditions satisfied:
   - The lower bound problem on M defined as (22) is infeasible.
   - $\beta(M) > 0$.

For starting DCA on a partition M, we need to find an initial point. A good initial point will lead a fast convergence and a global optimal solution on M. When the compact constraint set $\mathcal{C} \cap M$ is not empty, the existence of such a good initial point is ensured since a global optimal solution on M leads DCA immediately stopped at this solution point. However, estimating a good initial point is hard. We propose to solve

the SDP lower bound estimation problem (22) whose optimal solution $(\bar{\mathbf{x}}, \bar{W})$ will be used as initial point to start DCA on the partition set M.

Now, we can describe the new DCA and B&B method for QMIFP as:

DCA-B&B for QMIFP

Initialization:
Start with $M_0 = [\underline{\mathbf{x}}, \bar{\mathbf{x}}]$.
Solving the SDP problem (22) to find an initial point $(\mathbf{x}^0, W^0)$ and $\beta(M_0)$.
Define ϵ_1 as a sufficiently small positive number.
Let iteration number $k = 0$ and $\mathcal{S} = \{M_0\}$.
If $f(\mathbf{x}^0, W^0) \leq \epsilon_1$, then STOP algorithm
QMIFP is feasible and $(\mathbf{x}^0, W^0)$ is a feasible solution.
otherwise, goto Step1.

Step1:
If $\mathcal{S}$ is empty set, then terminate: QMIFP is infeasible.
Otherwise, find a partition set $M_k = [p, q] \in \mathcal{S}$ with the smallest $\beta(M_k)$.
Define $(\mathbf{x}^*, W^*)$ to be the optimal solution corresponding to $\beta(M_k)$.
Start DCA with initial point $(\mathbf{x}^*, W^*)$ to problem (21), its computed solution is denoted as $(\mathbf{x}^k, W^k)$.
If $\alpha(M) := f(\mathbf{x}^k, W^k) \leq \epsilon_1$, then STOP algorithm
QMIFP is feasible and $(\mathbf{x}^k, W^k)$ is a feasible solution.
otherwise, goto Step2.

Step2:
- $(i^*, j^*) \in argmax\{|w^*_{ij} - x^*_i x^*_j| : \forall i, j = 1, \ldots, N\}$.
- $i_{M_k} = argmax\{|p_{i^*} - q_{i^*}|, |p_{j^*} - q_{j^*}|\}$.
- $\lambda = (p_{i_{M_k}} + q_{i_{M_k}})/2$.
- Divide M_k into two smaller rectangles $M_{k,1} := M_k \cap \{p_{i_{M_k}} \leq x_{i_{M_k}} \leq \lambda\}$ and $M_{k,2} := M_k \cap \{\lambda \leq x_{i_{M_k}} \leq q_{i_{M_k}}\}$. goto Step3.

Step3:
For two partition sets $M_{k,1}$ and $M_{k,2}$, computing $\beta(M_{k,1})$ and $\beta(M_{k,2})$.
If $\beta(M_{k,1}) \leq \epsilon_1$ (resp. $\beta(M_{k,2}) \leq \epsilon_1$), then
Set $\mathcal{S} = \mathcal{S} \cup M_{k,1}$ (resp. $\mathcal{S} = \mathcal{S} \cup M_{k,2}$).
Endif
Set $\mathcal{S} = \mathcal{S} \setminus M_k$.
Set $k = k + 1$ goto Step1.

8 Computational Experiments

In this section, we present the computational experience for the proposed DCA and B&B Algorithm. The new algorithm were implemented in MATLAB Version2008A and tested on PC (Vista, Intel P8400, CPU 2GHz with 2G of memory). The software YALMIP [43] was incorporated in the subroutines to create LMI constraints and to solve SDP subproblems. YALMIP provides an advanced MATLAB interface for us-

ing several numerical optimization programs. In our tests, we try to use 4 well-known SDP solvers SDPA [39] by Kojima, SeDuMi [42] by Sturm, CSDP [41] by Borchers and DSDP [40] by Benson and Ye. They are all supported by YALMIP. Our computational results were compared with the commercial software PENBMI version 2.1 [38] developed by Michal Kočvara and Michael Stingl. This software combines the exterior penalty and interior barrier methods with the Augmented Lagrangian method. In our knowledge, it is one of the best BMI local solvers nowadays. Now, it has been integrated in the commercial optimization software TOMLAB. YALMIP has also given a interface to this solver.

Our test data were randomly generated. We use the method of the code "ranbmi.m" given in PENBMI for generating random BMI instances. Q_i and Q_{ij} were symmetric random $k \times k$ sparse matrices with a given density *dens* (i.e. with approximately $dens \times k \times k$ nonzero entries) whose elements are normally distributed, with mean 0 and variance 1. The lower and upper bounds of the variables x and y are restricted to the interval $[-10, 10]$.

The parameters for generating random test data are presented in Table 3.

Table 3 Parameters for generating random test data

parameter	value	description
n	any positive integer	dimension of the variable $\mathbf{x}$
m	any positive integer	dimension of the variable $\mathbf{y}$
k	any positive integer	Q_i and Q_{ij} are matrices of dimension $k \times k$
dens	a value in $]0, 1]$	density of sparse matrix for Q_i and Q_{ij}

As we have introduced in section 2, any BMIFP can be easily converted into a QMIFP. Therefore, all BMIFP test problems will be transformed into QMIFPs. A QMI constraint as

$$Q_0 + \sum_{i=1}^{N} x_i Q_i + \sum_{i,j=1}^{N} x_i x_j Q_{ij} \preceq 0$$

is defined by the matrices $Q_i, i = 0, \ldots, N$ and $Q_{ij}, i, j = 1, \ldots, N$. Since $Q_{ij} = Q_{ji}$, we only need to have $Q_i, i = 0, \ldots, N$ and $Q_{ij}, i = 1, \ldots, N; j = i, \ldots, N$ been generated. In our software, we defined a special QMI structure object whose fields were summarized in Table 4.

Table 4 QMI structure

field	type	description
$Q0$	matrix	a $k \times k$ matrix for Q_0
QI	list	a list of nonzero matrices for $Q_i, i = 1, \ldots, N$
$IdxQI.N$	integer	the number of total elements in the list QI
$IdxQI.IDX$	list	a list of index numbers for QI
QIJ	list	a list of nonzero matrices for $Q_{ij}, i = 1, \ldots, N; j = i, \ldots, N$
$IdxQIJ.N$	integer	the number of total elements in the list QIJ
$IdxQIJ.IDX$	matrix	the (i, j)-element denotes the index number of Q_{ij} in the list QIJ

Remark 6 For saving all nonzero matrices of $Q_i, i = 1, \ldots, N$, we use the fields QI $IdxQI.N$ and $IdxQI.IDX$. The list QI saves all nonzero matrices of $Q_i, i = 1, \ldots, N$, and $IdxQI.N$ denotes the total number of nonzero matrices. The list $IdxQI.IDX$ saves i for the nonzero matrices Q_i and 0 for zero matrices. For instance, suppose Q_1 and Q_3 are both nonzero matrices, and $Q_2 = 0$. For this case, $QI = \{Q_1, Q_3\}$, $IdxQI.N = 2$ and $IdxQI.IDX = \{1, 0, 3\}$. Using these three fields (QI, $IdxQI.N$ and $IdxQI.IDX$), we can easily access any matrix Q_i with a given index i form 1 to N. The fields (QIJ, $IdxQIJ.N$ and $IdxQIJ.IDX$) are analogue to the fields (QI, $IdxQI.N$ and $IdxQI.IDX$). The only difference is that, $IdxQIJ.IDX$ is a 1-D list, but $IdxQIJ.IDX$ is a 2-D matrix whose (i, j)-element denotes the index number of Q_{ij} in the list QIJ. Obviously, $IdxQIJ.IDX$ must be a symmetric matrix since $Q_{ij} = Q_{ji}$ for all i and j for 1 to N. With our QMI structure, we can easily define a QMI constraint and access all matrices Q_i and Q_{ij} from this structure.

The parameters for the new DCA-B&B algorithm are summarized in Table 5.
Now, let us present some numerical experiments and their results.

Table 5 Parameters for the DCA and DCA-B&B Algorithms

parameter	value	description
ϵ_1	10^{-5}	computational zero (for DCA and DCA-B&B)
ϵ_2	10^{-3}	relative error - stopping criterion for DCA

First, we will compare the performance of DCA incorporated with different SDP solvers (SDPA, SeDuMi, CSDP and DSDP). A fast SDP solver is very important since DCA needs solving a SDP subproblem at each iteration. In general, it is known that SDPA is a faster SDP solver for medium and large-scale problems than SeDuMi and other two solvers. See Mittelmann [44] for a comparison of the performance of various solvers (including SDPA, SeDuMi, CSDP and DSDP) on the SDPLIB test set.

In order to find the fastest SDP solver for combination with DCA, we randomly generate 10 BMIFPs with fixed parameters $n = 2$, $m = 3$, $k = 3$ and $dens = 0.1$, then solving them via DCA. The consuming time were summarized in Table 6.

Table 6 shows SDPA should be the fastest SDP solver, incorporated with our DCA, than the other SDP solvers compared in this paper. We can order the tested SDP solvers in increasing order of the cost of computational time as SDPA, SeDuMi, CSDP and DSDP. The columns *feas* denote the feasibility of BMIFPs. T means DCA has found a feasible solution by DCA, while F means no feasible solution has been found by DCA within 30 iterations. For these problems, they are probably infeasible, since the objective values are always far from zero during the iterations. To guarantee the infeasibility, we should use DCA-B&B algorithm. Note that these SDP solvers for solving a given SDP subproblem do not often give a same optimal solution, but give the same objective value. Because one SDP subproblem is a convex programming whose optimal objective value is unique, but its optimal solution is not unique when the problem is not a strictly convex one. From the result in Table 6, we prefer to choose the fastest SDP solver - SDPA for the further tests.

Table 6 Performance of DCA with SDPA, SeDuMi, CSDP and DSDP

Problem	SDPA		SeDuMi		CSDP		DSDP	
	time	feas	time	feas	time	feas	time	feas
$P1$	0.148	T	1.203	T	0.960	T	0.736	T
$P2*$	1.158	F	2.434	F	3.887	F	2.120	F
$P3*$	1.169	F	2.464	F	4.154	F	2.180	F
$P4$	0.155	T	0.250	T	0.951	T	0.433	T
$P5$	0.439	T	0.688	T	1.153	T	0.677	T
$P6$	0.192	T	0.260	T	0.546	T	0.291	T
$P7$	0.369	T	0.240	T	0.822	T	1.314	T
$P8*$	1.175	F	0.278	F	4.072	F	1.685	F
$P9$	0.619	T	1.019	T	1.753	T	1.264	T
$P10$	0.233	T	0.313	T	0.796	T	0.324	T
Average time	0.566		0.915		1.909		1.102	

* algorithm stopped after 30 iterations of DCA.

Table 7 gives some comparison results between our algorithm DCA and the software PENBMI on pseudo-randomly generated BMIFPs. For these test problems, the tolerance parameters ϵ_1 and ϵ_2 were fixed as in Table 5 and $dens = 0.1$. Each line of Table 5 consists of numerical results for a set of 20 feasible BMIFPs randomly generated with a given dimension m, n and k. the variables x and y are restricted to the interval $[-10, 10]$ and all matrices Q_i are of order from 3×3 to 100×100. The average iterations and the average CPU time for the DCA and the PENBMI were presented and compared for each set of problems.

Table 7 Performance comparison between the DCA and PENBMI for some sets of randomly generated feasible BMIFPs with different dimensions

Problem	Dimension			DCA		PENBMI	
	n	m	k	ave.iter	ave.time	ave.iter	ave.time
Pset1	1	1	3	1	0.1580	14.6	0.8136
Pset2	2	3	5	1.2	0.1668	15.6	0.8442
Pset3	8	10	10	1.3	1.316	18.6	1.815
Pset4	10	10	40	1.5	1.573	19.2	2.125
Pset5	10	10	60	1.2	2.667	20.2	3.243
Pset6	10	10	70	1.2	3.848	16.8	4.029
Pset7	10	10	100	1.5	7.816	18.2	10.249
Pset8	15	10	30	2.1	2.909	20.5	3.677
Pset9	15	10	60	2.8	5.328	20.5	6.583
Pset10	15	10	80	1.6	9.125	19.5	22.800
Pset11	20	15	20	1.3	5.869	20.5	2.362
Pset12	20	15	60	2.5	25.490	19.8	35.841
Pset13	30	30	10	1.2	27.17	21.5	5.287
Pset14	40	30	6	1.5	30.492	18.5	7.596
Pset15	50	50	10	1.2	50.585	18.2	13.685

Each problem set contains 20 randomly generated problems with a given dimension m,n,k.

Comparing the numerical results in Table 7, our DCA method needs only 2-3 iterations in average for finding the first feasible solution, while the PENBMI often needs 18-23 iterations. Since the objective of the BMIFP is to find a feasible solution if it exist, therefore, the computed feasible solution found by DCA and PENBMI could be different. In our tests, we need only to compare the computational time for finding

the first feasible solution by these two methods. Concerning the problems of dimension $n + m \leq 35$ and $k \leq 100$, DCA is often faster than PENBMI. Moreover, the total number of iterations is stable between 1 and 2.8 in average.

However, we also noticed that, DCA seems more expensive for solving the set of problems $Pset12$ - $Pset15$ than PENBMI. This is probably due the fact that DCA will solve a SDP subproblem with $m+n+\frac{(m+n)(m+n+1)}{2}$ variables with a LMI constraint of order $[3(m+n)+k+1] \times [3(m+n)+k+1]$ at each iteration. However, under the present level of technology, the SDP solvers can not solved so efficiently a large-scale SDP problem. Therefore, we proposed the algorithm (DCA2) in which a SDP subproblem is not required to be solved entirely. This new approach could be more efficient to handle the large-scale problems. Some numerical results is presented in the Table (9).

Table 8 Performance comparison between DCA1 and DCA2

Problem	Dimension			DCA1		DCA2	
	n	m	k	ave.iter	ave.time	ave.iter	ave.time
Pset1	1	1	3	1	0.1580	5.8	0.8210
Pset2	2	3	5	1.2	0.1668	5.3	0.662
Pset3	8	10	10	1.3	1.316	5.5	1.938
Pset4	10	10	40	1.5	1.573	6.7	2.982
Pset5	10	10	60	1.2	2.667	5.2	2.332
Pset6	10	10	70	1.2	3.848	5.0	4.398
Pset7	10	10	100	1.5	7.816	6.3	10.210
Pset8	15	10	30	2.1	2.909	7.1	11.897
Pset9	15	10	60	2.8	5.328	8.8	15.199
Pset10	15	10	80	1.6	9.125	5.6	20.928
Pset11	20	15	20	1.3	5.869	5.1	12.360
Pset12	20	15	60	2.5	25.490	6.8	20.868
Pset13	30	30	10	1.2	27.17	5.5	25.836
Pset14	40	30	6	1.5	30.492	6.5	29.630
Pset15	50	50	10	1.2	50.585	5.1	38.331

Each problem set contains 20 problems.

We can find in the result table 8 that, DCA2 is more faster than DCA1 for large-scale problems (such as Pset15), however, for small sized problems, DCA1 is faster since it needs less iteration, and solving a small sized SDP problem is not expensive. Furthermore, the number of iterations of DCA2 is also stable between $5.3 - 8.8$ in average. Therefore, we can conclude that DCA1 is recommended for small sized problems and DCA2 is suggested for large-scale ones.

During our tests, we also finds some random problem for which DCA converges without finding a feasible solution, i.e. the objective value is still positive. For these examples, we test our DCA-BB algorithm and some of them have been determined as feasible problem. We summarized some results in table 9

We can see that DCA-B&B can find a feasible solution of BMI for these examples which have not been solved directly by applying DCA and PENBMI. We must emphasize that the efficiency of DCA-B&B depends on the quality of the lower bound. A tighter and cheaper lower bound will lead to a fast convergence of this method.

Table 9 Comparison results among DCA, PENBMI and DCA-B&B

Prob	size			DCA			PENBMI			DCA-B&B		
	m	n	k	obj	iter	time	obj	iter	time	obj	node	time
P1	3	5	3	0.035	1	0.178	1.275	19	1.028	1.12E-6	30	6.818
P2	5	8	5	0.088	2	0.339	1.287	20	1.317	3.31E-6	26	28.991
P3	8	10	10	0.092	1	1.512	0.008	20	2.216	5.19E-6	69	268.025
P4	10	10	40	0.011	3	2.025	2.105	19	3.276	8.75E-6	33	236.102
P5	10	10	60	0.059	3	3.311	0.211	20	3.288	9.21E-6	82	356.811

9 Conclusion

In this paper, we propose to use DC programming approaches (DCA) for solving the Bilinear Matrix Inequality (BMI) Feasibility Problems and the Quadratic Matrix Inequality (QMI) Feasibility Problems. We studied the equivalent DC programming formulation and solved by the proposed algorithms (DCA1, DCA2 and DCA-B&B). DCA1 requires solving a SDP problem at each iteration. DCA2 is a improved method from DCA1 with considering the partial solution strategy which is suitable for solving large-scale problems. The DCA-B&B can be used for guaranteeing the feasibility of a given BMI or QMI. The comparison numerical results show that our proposed methods are faster than PENBMI for some sizes of problems.

There are some other topics involving BMI and QMI such as the Bilinear Matrix Inequality Eigenvalue Problem (BMIEP), and the optimization problems with BMI and QMI constrains. They are more difficult to be solved than the feasibility problems. Researches in these directions are currently in progress; their results will be reported subsequently.

References

1. Tuan H.D., Hosoe S., Tuy H.: D.C. optimization approach to robust controls: Feasibility problems, IEEE Transactions on Automatic Control, Vol. 45, 1903-1909, (2000)
2. Goh. K.C., Safonov M.G., Papavassilopoulos G.P.: Global optimization for the biaffine matrix inequality problem. Journal of Global Optimization, Vol. 7, 365-380, (1995)
3. Goh K.C., Safonov M.G., and Ly J. H.: Robust synthesis via bilinear matrix inequalities. International Journal of Robust and Nonlinear Control, Vol. 6, 1079-1095, (1996)
4. Sherali H.D., Alameddine A.R.: A new reformulation-linearization technique for bilinear programming problems. Journal of Global Optimization, Vol. 2, 379-410, (1992)
5. Toker O., Özbay H.: On the NP-hardness of solving bilinear matrix inequalities and simultaneous stabilization with static output feedback. American Control Conference, Seattle, WA, (1995)
6. Safonov M.G., Goh K.C., Ly J.H.: Control system synthesis via bilinear matrix inequalities. In Proceedings of the American Control Conference, Baltimore, MD, June (1994)
7. VanAntwerp J.G.: Globally optimal robust control for systems with time-varying nonlinear perturbations. Master thesis, University of Illinois at Urbana-Champaign, Urbana, IL, (1997)
8. Fujioka H., Hoshijima K.: Bounds for the BMI eingenvalue problem - a good lower bound and a cheap upper bound. Transactions of the Society of Instrument and Control Engineers, Vol. 33, 616-621, (1997)
9. Fukuda M., Kojima M.: Branch-and-Cut Algorithms for the Bilinear Matrix Inequality Eigenvalue Problem. Computational Optimization and Applications, Vol. 19, Issue 1, 79-105, (2001)

10. Kawanishi M., Sugie T., Kanki H.: BMI global optimization based on Branch-and-Bound method taking account of the property of local minima. In Proceedings of the Conference on Decision and Control, San Diego, CA, December (1997)
11. Takano S., Watanabe T., Yasuda K.: Branch-and-Bound technique for global solution of BMI. Transactions of the Society of Instrument and Control Engineers, Vol. 33, 701-708, (1997)
12. Liu S.M., Papavassilopoulos G.P.: Numerical experience with parallel algorithms for solving the BMI problem. 13th Triennial World Congress of IFAC, San Francisco, CA, July (1996)
13. Mesbahi M., Papavassilopoulos G.P.: A cone programming approach to the bilinear matrix inequality problem and its geometry. Mathematical Programming, Vol. 77, 247-272, (1997)
14. Floudas C. A., Visweswaran V.: A primal-relaxed dual global optimization approach. Journal of Optimization Theory and Applications, Vol. 78, 187-225, (1993)
15. Beran E.B., Vandenberghe L., Boyd S.: A global BMI algorithm based on the generalized Benders decomposition. In Proceedings of the European Control Conference, Brussels, Belgium, July (1997)
16. Tuan H.D., Apkarian P., Nakashima Y.: A new Lagrangian dual global optimization algorithm for solving bilinear matrix inequalities. International Journal of Robust and Nonlinear Control, Vol. 10, 561-578, (2000)
17. Pham Dinh T., Le Thi H.A., DC Programming. Theory, Algorithms, Applications: The State of the Art. First International Whorkshop on Global Constrained Optimization and Constraint Satisfaction, Nice, October 2-4, (2002)
18. Pham Dinh T., Le Thi H.A., Convex analysis approach to D.C. programming: Theory, Algorithms and Applications. Acta Mathematica Vietnamica, Vol.22 No.1, 289-355, (1997)
19. Pham Dinh T., Le Thi H.A.: The DC programming and DCA Revisited with DC Models of Real World Nonconvex Optimization Problems. Annals of Operations Research, Vol.133, 23-46, (2005)
20. Le Thi H.A., Pham Dinh T.: large-scale Molecular Optimization From Distance Matrices by a D.C. Optimization Approach. SIAM Journal on Optimization, Vol. 4(1), 77-116, (2003).
21. Le Thi H.A., Pham Dinh T., Le Dung M.: Exact penalty in d.c. programming. Vietnam Journal of Mathematics, Vol. 27(2), 169-178, (1999)
22. Le Thi H.A., Pham Dinh T.: Solving a class of linearly constrained indefinite quadratic problems by DC Algorithms. Journal of Global Optimization, Vol. 11, 253-285, (1997).
23. Le Thi H.A., Pham Dinh T.: A continuous approach for large-scale constrained quadratic zero-one programming. (In honor of Professor ELSTER, Founder of the Journal Optimization), Optimization Vol. 45(3), 1-28, (2001)
24. Pham Dinh T., Le Thi H.A.: DC optimization algorithms for solving the trust region subproblem, SIAM J.Optimization, Vol. 8, 476-507, (1998)
25. Le Thi H.A.: Solving large-scale molecular distance geometry problems by a smoothing technique via the gaussian transform and d.c. programming, Journal of Global Optimization, Vol. 27(4), 375-397, (2003)
26. Le Thi H.A.: An efficient algorithm for globally minimizing a quadratic function under convex quadratic constraints. Mathematical Programming, Ser. A, Vol. 87(3), 401-426, (2000)
27. Le Thi H.A., Pham Dinh T., François A.: Combining DCA and Interior Point Techniques for large-scale Nonconvex Quadratic Programming. Optimization Methods & Software, Vol. 23(4), 609-629, (2008)
28. Niu Y.S., Pham Dinh T.: A DC Programming Approach for Mixed-Integer Linear Programs. Computation and Optimization in Information Systems and Management Sciences, 244-253, Book Series: Communications in Computer and Information Science, Springer Berlin Heidelberg, (2008)
29. Horst R.: D.C. Optimization: Theory, Methods and Algorithms, in: R. Horst and P.M. Pardalos (eds.), Handbook of Global Optimization, 149-216. Kluwer Academic Publishers, Dordrecht, The Netherlands. (1995)
30. Horst R., Thoai N.V.: DC Programming: Overview. Journal of Optimization Theory and Applications, Vol. 103, 1-43, Springer Netherlands, (1999)
31. Horst R., Pardalos P.M., Thoai N.V.: Introduction to Global Optimization - Second Edition. Kluwer Academic Publishers, Netherlands, (2000)
32. Rockafellar R.T.: Convex Analysis. Princeton University Press, N.J., (1970)
33. Hiriart Urruty J.B., Lemaréchal C.: Convex Analysis and Minimization Algorithms. Springer, Berlin, Heidelberg (1993)

34. Horst R.: D.C. Optimization: Theory, Methods and Algorithms, in: R. Horst and P.M. Pardalos (eds.), Handbook of Global Optimization, 149-216. Kluwer Academic Publishers, Dordrecht, The Netherlands. (1995)
35. Horst R., Thoai N.V.: DC Programming: Overview. Journal of Optimization Theory and Applications, Vol. 103, 1-43, Springer Netherlands, (1999)
36. Horst R., Pardalos P.M., Thoai N.V.: Introduction to Global Optimization - Second Edition. Kluwer Academic Publishers, Netherlands, (2000)
37. Wolkowicz H., Saigal R., Vandenberghe L.: Handbook of Semidefinite Programming - Theory, Algorithms, and Applications. Kluwer Academic Publishers, USA, (2000)
38. Kočvara M., Stingl M.: PENBMI User's Guide (Version 2.1). February 16, (2006)
39. K. Fujisawa, M. Kojima and K. Nakata: SDPA (SemiDefinite Programming Algorithm) - user's manual - version 6.20. Research Report B-359, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, Tokyo, Japan, January 2005, revised May (2005). Available at `http://sdpa.indsys.chuo-u.ac.jp/sdpa/download.html`
40. Benson S.T., Ye Y.Y.: DSDP: A complete description of the algorithm and a proof of convergence can be found in Solving Large-Scale Sparse Semidefinite Programs for Combinatorial Optimization, SIAM Journal on Optimization, 10(2), pp. 443-461, (2000). Available at `http://www.mcs.anl.gov/hs/software/DSDP/`
41. B. Borchers, CSDP: a C library for semidefinite programming, Department of Mathematics, New Mexico Institute of Mining and Technology, Socorro, NM, November (1998). Available at `https://projects.coin-or.org/Csdp/`
42. Sturm J.F.: SeDuMi 1.2: a MATLAB toolbox for optimization over symmetric cones. Department of Quantitative Economics, Maastricht University, Maastricht, The Netherlands, August (1998). Available at `http://sedumi.ie.lehigh.edu/`
43. Löfberg J.: YALMIP: A Toolbox for Modeling and Optimization in MATLAB. In Proceedings of the CACSD Conference, Taipei, Taiwan, (2004). `http://control.ee.ethz.ch/~joloef/wiki/pmwiki.php`
44. Mittelmann H.D.: Several SDP-codes on problems from SDPLIB. `http://plato.asu.edu/ftp/sdplib.html`
45. MATLAB R2007a: Documentation and User Guides. `http://www.mathworks.com/`

Appendices

Annexe A

Program Codes, Prototypes and Softwares

In this chapter, we'd like to briefly present some parts of our program codes, prototypes and softwares. The full program codes developed during my Ph.D researches will cost thousands pages. So we only pick out a very small part as illustrative examples in Annexes. The DCA and DCA-BB prototype codes are also presented.

A.1 GUI Interface

Firstly, we give a short presentation to the GUI software that I designed using E-language and C++. This tool can be used as a common interface to invoke our DCA solvers for solving different types of DC programs. Here is a screenshot of the software :

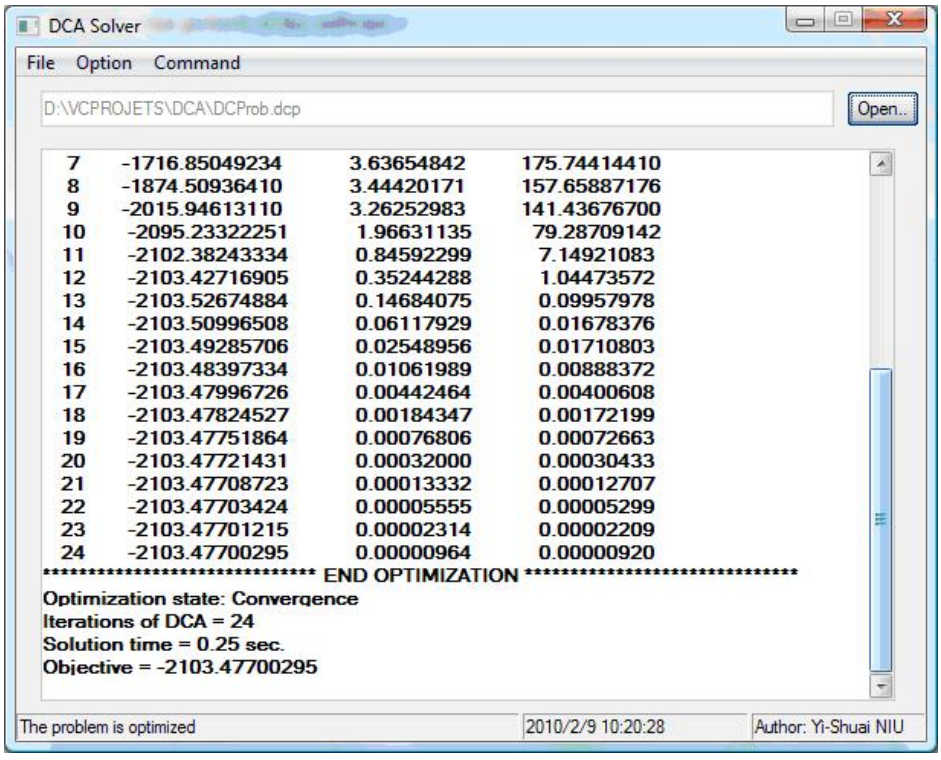


Fig. A.1 – DCA GUI Interface

This interface provides a simple and convenient platform for both users and programmers.

For the users, it is very easy to use this software. You need only pressing the button "open" or "Ctrl+O" to choose a data file (a special file format with file extension ".dcp" that I designed for DC program), then the software can recognize the data file format and read the date from files. After reading a problem date file, some information about the corresponding DC problem will be shown in the text field. For optimizing the problem, you can choose in the menu "Command - Optimize" to run the solution procedure. Before running the optimization command, you are able to use "Option - Setting" for modifying your DCA's parameters, such as Max iterations, predefined tolerances, and other options (as verbose options etc). The computational results will appear at the end of the optimization in the text field. You can save your results and other needed information into a file via the menu "file - save as".

On the other hand, it is convenient for the programmers to integrate their solution codes to this interface. You can implement your solution codes using any programming language and compile them as a ".dll" library file. Here, you need exporting some common predefined structured functions in your dll in order to give the GUI interface a way to call your solvers. These export functions including : a function to recognize the data file format : "IsSupportFormat" ; a function to read the problem from data file : "ReadProbFile" ; a function to initialize the solver : "InitSolver" ; a function to modify and define the parameters : "DefParam" ; and a callback function to invoke the dll file for solving your optimization problem : "Optimize". So far, this interface is only supporting our DCA Algorithms, however this design conception will create infinitely many possibilities to integrate with different kinds of solvers and data file formats. It is possible to be continuously developed for conveniently handling other algorithms in the future.

A.2 Read and Write MPS and LP files in MATLAB

In this section, we present our implementation of MATLAB mex codes in C language for reading and writing .mps and .lp data file formats (supported by CPLEX [104]) via MATLAB. In our knowledge, there is only two commercial softwares "MOSEK" [106] and "TOMLAB" [107] can read and write these file formats on MATLAB platform before 2008 (Since 2009, the latest CPLEX 12 begins integrating with MATLAB and provides a Matlab Toolbox for conveniently using CPLEX via MATLAB platform. This Toolbox provides also API callback functions for reading and writing MPS, LP files in MATLAB). However, All these softwares are commercial softwares which have only trial versions for free users. In our researches, we often need solving large-scale problems which can not be handled by the trial one. Therefore, we developed a powerful MATLAB mex code for transforming the problem data between

MATLAB format and .mps or .lp file format at the beginning of 2008 (at that moment, the CPLEX 11 is equipped in our Library LMI). So I decide to develop this tool which is not only necessary to my Ph.D. researches, but also a remarkable contribution to the optimization field. The code is designed for reading and writing the following QCQP (continuous and mixed-integer) problem type :

$$\min \quad f(x) := \frac{1}{2}x^T Hx + f^T x$$

subject to :

$$Ax \leq (or =)b; lb \leq x \leq ub$$

$$x^T Q_i x + L_i^T x \leq r_i, i = 1, \ldots, nQC$$

The variable types are available as : $\{'C' = continuous|'B' = binary|'I' = integer|'S' = semi - continuous|'N' = semi - integer\}$. For more information about the variable types, the reader is refereed to [104].

MPS (Mathematical Programming System) (.mps) is a file format for presenting and archiving linear programming (LP) and mixed integer programs. The format was named after an early IBM LP product and has emerged as a de facto standard ASCII medium among most of the commercial LP solvers. Essentially all commercial LP solvers accept this format, and it is also accepted by the open-source COIN-OR system [108]. For more information about this format, the reader is referred to http://lpsolve.sourceforge.net/5.5/mps-format.htm.

The CPLEX LP format (.lp) is provided as an input alternative to the MPS file format. An LP format file may be easier to generate than an MPS file if your problem already exists in an algebraic format or if you have an application that generates the problem file more readily in algebraic format (such as a C application). for more details about this format, you can read this web http://lpsolve.sourceforge.net/5.5/CPLEX-format.htm.

A.2.1 Read .mps and .lp file to MATLAB

Here, we give some essential parts of our MATLAB mex code for reading problems from .lp and .mps files :

```
/* ------------------------------------------------------------------------
 * file:        CPXreadprob.c
 * Project:     MATLAB MEX interface for reading CPLEX supported problems
 *              (as lp,mps etc)
 *
 * Purpose:     Invoke the CPLEX callable lib.
 *
```

```
 * Authors:     Yi-Shuai NIU
 * Contact:     INSA de Rouen
 *              niuys@insa-rouen.fr
 *
 * History: date: 2008.01.20  ver1.1
 * ------------------------------------------------------------------------
 *
 * Notes        This file requires CPLEX (9.0 or higher version)
 *              and MATLAB (6.0 or higher) to compile as CPXreadprob.mexw32
 *
 *              CPLEX    http://www.ilog.com
 *              MATLAB   http://www.mathworks.com
 *
 * (C) Jan. 20 2008 by Yi-Shuai NIU
 * Revision: 1.1
 * All rights reserved.
 */------------------------------------------------------------------------
/* Comments:
  Matlab interface to CPLEX 9.0 or higher version for the following
  optimization problem:

     min    0.5*x'*H*x + f'*x
      x
     s.t.:  A x {'<=' | '='} b
            x' * QC(i).Q * x + QC(i).L * x <= QC(i).r,  i=1,...,nQC
            LB <= x <= UB
            x(i) is {'C' = continuous | 'B' = binary | 'I' = integer |
             'S' = semi-continuous | 'N' = semi-integer}, i=1,...,n

  The calling syntax is:
  [STAT,H,f,A,b,INDEQ,QC,LB,UB,VARTYPE] = CPXwriteprob(FILENAME,DISP)

  Default: DISP = 0 no information
  DISP = 1 show all informations
  DISP = 2 show important informations
*/

/* MATLAB declarations. */
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <matrix.h>
#include "mex.h"

/* CPLEX declarations.  */
```

```
#include "cplex.h"


/* CPLEX env */
static CPXENVptr env = NULL;
static CPXLPptr  lp = NULL;
static CPXFILEptr LogFile = NULL;


/* MEX Input Arguments */
enum {NAME_IN_POS,MAX_NUM_IN_ARG};

#define MIN_NUM_IN_ARG     1
#define MAX_NUM_IN_ARG     2

#define NAME_IN  prhs[0]
#define DISP_IN  prhs[1]


/* MEX Output Arguments */
enum {STAT_OUT_POS, H_OUT_POS, F_OUT_POS, A_OUT_POS, B_OUT_POS,
INDEQ_OUT_POS, QC_OUT_POS, LB_OUT_POS, UB_OUT_POS,
VARTYPE_OUT_POS, MAX_NUM_OUT_ARG};

#define MIN_NUM_OUT_ARG      10
#define MAX_NUM_OUT_ARG                10

#define STAT_OUT      plhs[0]
#define H_OUT         plhs[1]
#define F_OUT         plhs[2]
#define A_OUT         plhs[3]
#define B_OUT         plhs[4]
#define INDEQ_OUT     plhs[5]
#define QC_OUT        plhs[6]
#define LB_OUT        plhs[7]
#define UB_OUT        plhs[8]
#define VARTYPE_OUT   plhs[9]

/* Global variables */
//DOUBLE PRECISION CORRESPONDANCES OF ALL ARGUMENTS
char *FILENAME;
double *disp;


//DOUBLE PRECISION CORRESPONDANCE OF THE OUTPUT
```

```
//objective
double         *H = NULL;
double   *F = NULL;

//linear constraints
double         *A = NULL;
double   *b = NULL;
double   *INDEQ = NULL;
char     *sense = NULL;

//variable types
char         *ctype = NULL;

//bounds of variables
double   *LB = NULL;
double   *UB = NULL;

//Quadratic constraints
double   *QC = NULL;


//size of problem
int nrows=0;
int ncols=0;
int nvars=0;

//problem type
int probtype;
// -1 no type, 0 LP, 1 MILP, 3 FIXEDMILP, 5 QP,
// 7 MIQP, 8 FIXEDMIQP, 10 QCP, 11 MIQCP


/* This hack is because Matlab R14 can crash on Linux due to the call to
   mexErrMsgTxt() */
#define TROUBLE_mexErrMsgTxt(A)    mexPrintf(A); mexPrintf("\n"); return
/* Here is the original version
#define TROUBLE_mexErrMsgTxt(A)    mexErrMsgTxt(A)
*/

/* Functions declarations */
static int
readprob       (CPXENVptr env, CPXLPptr lp, mxArray *plhs[]);

static void
free_and_null  (char **ptr),
```

```
usage          (char *progname),
printVF        (double *v, int nvars),
printVI        (int *v, int nvars),
printMF        (double *matrix, int nrows, int ncols),
printMI        (int *matrix, int nrows, int ncols),
GetMatrix      (double *M, int *cmatbeg, int *cmatind,
              double *cmatval, int cmatsz, int nrows, int ncols);

// THE GATEWAY ROUTINE
void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                                const mxArray *prhs[])
{
    int           status = 0;
    int           i;
    double                  *statout;

//CONNECT TO THE INPUTS TROUGHT POINTER ASSIGNMENTS
//        disp = new int;
    if (nrhs==2)
    {
        disp = mxGetPr(DISP_IN);
    }
    if (nrhs<2)
        *disp = 0;

    i = mxGetNumberOfElements(NAME_IN)+1;
    /* Allocate enough memory to hold the converted string. */
    FILENAME = (char *)mxCalloc(i, sizeof (char));

    /* Copy the string data */
    if (mxGetString(NAME_IN, FILENAME, i) != 0)
    {
        TROUBLE_mexErrMsgTxt("Could not convert string data.");
    }


    /* 1- Define CPLEX variables  */
    CPXENVptr     env = NULL;
    CPXLPptr      lp = NULL;

    /* 2- Initialize CPLEX environment */

    env = CPXopenCPLEX (&status);
    if ( env == NULL )
    {
```

```
    char  errmsg[1024];
    fprintf (stderr, "Could not open CPLEX environment.\n");
    CPXgeterrorstring (env, status, errmsg);
    fprintf (stderr, "%s", errmsg);
    goto TERMINATE;
}

/* -----use some options here------*/
/* Turn on output to the screen */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_OFF);
if ( status )
{
    fprintf (stderr,
             "Fail to turn on screen indicator, error %d.\n", status);
    goto TERMINATE;
}


/* 3 - Construct problem */

lp = CPXcreateprob (env, &status, "lpex1");
if ( lp == NULL )
{
    fprintf (stderr, "Failed to create LP.\n");
    goto TERMINATE;
}

/* 4- Openfile */
status = CPXreadcopyprob(env, lp, FILENAME, NULL);
if (status==0)
{
    printf("YES\n");
}
else
{
    printf("NO\n");
    goto TERMINATE;
}

/* 5- Read data */
status = readprob (env, lp, plhs);
if ( status )
{
    fprintf (stderr, "Failed to populate problem.\n");
```

```
        goto TERMINATE;
    }


//Allocate memory to output variables
    STAT_OUT = mxCreateDoubleMatrix(1, 1, mxREAL);

//pointer to the memory space
    statout = mxGetPr(STAT_OUT);

    *statout = status;


TERMINATE:

    /* Free up the problem as allocated by CPXcreateprob, if necessary */

    if ( lp != NULL )
    {
        status = CPXfreeprob (env, &lp);
        if ( status )
        {
            fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
        }
    }

    /* Free up the CPLEX environment, if necessary */

    if ( env != NULL )
    {
        status = CPXcloseCPLEX (&env);

        /* Note that CPXcloseCPLEX produces no output,
        so the only way to see the cause of the error is to use
        CPXgeterrorstring.  For other CPLEX routines, the errors will
        be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

        if ( status )
        {
            char  errmsg[1024];
            fprintf (stderr, "Could not close CPLEX environment.\n");
            CPXgeterrorstring (env, status, errmsg);
            fprintf (stderr, "%s", errmsg);
        }
    }
```

```
}
/* END OF THE GATEWAY ROUTINE */

/* ********************************************************************** */


/*********************************************************
/*  Reading Data from file
/*********************************************************/

static int
readprob (CPXENVptr env, CPXLPptr lp, mxArray *plhs[])
{
    // local variables
    int i,j;
    int status    = 0;

    int                 nnz=0;
    int      *cmatbeg = NULL;
    int      *cmatind = NULL;
    double   *cmatval = NULL;
    int                 cmatsz = 0;
    int                 cmatspace = 0;
    int                 surplus = 0;


    int objsen = 0; // CPX_MAX=1, CPX_MIN=-1


    int      *qmatbeg = NULL;
    int      *qmatind = NULL;
    double   *qmatval = NULL;
    int                 qmatsz = 0;
    int                 qmatspace = 0;

    int nEq = 0;
    int *indlist;


    // Analyse data
    // 1. get problem size
    nrows = CPXgetnumrows (env, lp);
    ncols = CPXgetnumcols (env, lp);
    nvars = ncols;
```

```
if (*disp == 1 || *disp == 2)
    printf("%d rows %d cols %d variables \n",nrows, ncols, nvars);


// 2. get optimization sense
objsen = CPXgetobjsen(env,lp);
if (*disp == 1 || *disp == 2)
{
    if (objsen ==1 )
        printf("minimization \n");
    else if (objsen == -1)
        printf("maximization\n");
    else
    {
        printf("failed\n");
        goto TERMINATE;
    }
}

// 3. get problem type
probtype = CPXgetprobtype (env, lp);
if (*disp == 1 || *disp == 2)
    printf("problem type: %d\n",probtype);

// change type using CPXchgprobtype(env,lp,CPXPROB_LP);


// 4. get obj
// 4.1 get linear part to F
F_OUT = mxCreateDoubleMatrix(nvars, 1, mxREAL);
F = mxGetPr(F_OUT);

status = CPXgetobj (env, lp, F, 0, nvars-1);
if (status)
{
    printf("get obj failed\n");
    goto TERMINATE;
}
else
{
    if (*disp == 1)
    {
        printf("----------Obj linear part F----------:\n");
        printVF(F,nvars);
    }
```

```
}

// 4.2 get quadratic part to H
if (probtype == CPXPROB_QP )
{
    /* status = CPXgetquad (env, lp, &nnz, qmatbeg, qmatind,
    qmatval, qmatspace, &surplus, 0,
    cur_numquad-1);
        */

    qmatbeg = (int *)malloc(ncols*sizeof(int));
    status = CPXgetquad (env, lp, &nnz, qmatbeg, NULL, NULL,
                         0, &surplus, 0, ncols - 1);
    if ( status != CPXERR_NEGATIVE_SURPLUS )
    {
        if ( status != 0 )
        { printf ("CPXgetcols allocate error, status: %d\n", status);
          goto TERMINATE;
        }
        printf("All columns in range [%d, %d] are empty.\n",
               0, (ncols - 1));
    }

    qmatsz   = -surplus;
    qmatind  = (int *) malloc ((1 + qmatsz)*sizeof(int));
    qmatval  = (double *) malloc ((1 + qmatsz)*sizeof(double));

    status = CPXgetquad (env, lp, &nnz, qmatbeg, qmatind, qmatval,
                         qmatsz, &surplus, 0, ncols - 1);
    if ( status )
    {
        printf("CPXgetcols failed, status = %d\n", status);
        goto TERMINATE;
    }
    //printVI(cmatbeg,ncols);
    //printVI(cmatind,cmatsz);
    //printVF(cmatval,cmatsz);

    H_OUT = mxCreateDoubleMatrix(nvars, nvars, mxREAL);
    H = mxGetPr(H_OUT);
    GetMatrix(H,qmatbeg,qmatind,qmatval,qmatsz,nvars,nvars);

    if (*disp == 1)
    {
        printf("----------obj's quadratic matrix H----------:\n");
```

```
            printMF(H,nvars,nvars);
        }

    }
    else
    {
        H_OUT = mxCreateDoubleMatrix(0, 0, mxREAL);
    }


    // 5. get constraints matrices
    cmatbeg = (int *)malloc(ncols*sizeof(int));
    status = CPXgetcols (env, lp, &nnz, cmatbeg, NULL, NULL,
                         0, &surplus, 0, ncols - 1);
    if ( status != CPXERR_NEGATIVE_SURPLUS )
    {
        if ( status != 0 )
        {
            printf ("CPXgetcols allocating error, status = %d\n", status);
            goto TERMINATE;
        }
        printf("All columns in range [%d, %d] are empty.\n",
               0, (ncols - 1));
    }

    cmatsz   = -surplus;
    cmatind  = (int *) malloc ((1 + cmatsz)*sizeof(int));
    cmatval  = (double *) malloc ((1 + cmatsz)*sizeof(double));

    status = CPXgetcols (env, lp, &nnz, cmatbeg, cmatind, cmatval,
                         cmatsz, &surplus, 0, ncols - 1);
    if ( status )
    {
        printf("CPXgetcols failed, status = %d\n", status);
        goto TERMINATE;
    }
    //printVI(cmatbeg,ncols);
    //printVI(cmatind,cmatsz);
    //printVF(cmatval,cmatsz);

    A_OUT = mxCreateDoubleMatrix(nrows, ncols, mxREAL);
    A = mxGetPr(A_OUT);
    GetMatrix(A,cmatbeg,cmatind,cmatval,cmatsz,nrows,ncols);

    if (*disp == 1)
```

```
{
    printf("----------constraints matrix----------:\n");
    printMF(A,nrows,ncols);
}


// 6- get rhs
B_OUT = mxCreateDoubleMatrix(nrows, 1, mxREAL);
b = mxGetPr(B_OUT);
status = CPXgetrhs (env, lp, b, 0, nrows-1);

if (status)
{
    printf("get rhs error\n");
    goto TERMINATE;
}
else
{
    if (*disp == 1)
    {
        printf("----------rhs----------:\n");
        printVF(b,nrows);
    }
}

// 7- constraint sense 'L'(<=), 'E'(==), 'G'(>=)
// convert all to 'L' and 'E'
sense = (char *)malloc(nrows*sizeof(char));
status = CPXgetsense (env, lp, sense, 0, nrows-1);
sense[nrows]='\0';
if (*disp == 1)
    printf("----------constraint sense----------:\n%s\n",sense);

indlist = (int *)malloc(nrows*sizeof(int));
for (i=0;i<nrows;i++)
{
    if (sense[i]=='G')
    {
        b[i] = -b[i];
        for (j=0;j<ncols;j++)
        {
            A[j*nrows+i] = -A[j*nrows+i];
        }
        sense[i] = 'L';
    }
```

```
        if (sense[i]=='R')
        {
            printf("R is not an available sense\n");
            goto TERMINATE;
        }
        if (sense[i]=='E')
        {
            indlist[nEq]=i+1; //Matlab index from 1
            nEq++;
        }
    }

    if (nEq!=0)
    {
        INDEQ_OUT = mxCreateDoubleMatrix(nEq, 1, mxREAL);
        INDEQ = mxGetPr(INDEQ_OUT);
        for (i=0;i<nEq;i++)
        {
            INDEQ[i]=indlist[i];
        }
    }
    else
    {
        INDEQ_OUT = mxCreateDoubleMatrix(0, 0, mxREAL);
    }


    // 8- get lower bound and upper bound
    LB_OUT = mxCreateDoubleMatrix(ncols, 1, mxREAL);
    LB = mxGetPr(LB_OUT);
    UB_OUT = mxCreateDoubleMatrix(ncols, 1, mxREAL);
    UB = mxGetPr(UB_OUT);
    status = CPXgetlb (env, lp, LB, 0, ncols-1);
    status = CPXgetub (env, lp, UB, 0, ncols-1);

    if (*disp == 1)
    {
        printf("----------lb----------:\n");
        printVF(LB,ncols);
        printf("----------ub----------:\n");
        printVF(UB,ncols);
    }


    // 9- get variable type
```

```
    /*        CPX_CONTINUOUS 'C' continuous variable
    CPX_BINARY 'B' binary variable
    CPX_INTEGER 'I' general integer variable
    CPX_SEMICONT 'S' semi-continuous variable
    CPX_SEMIINT 'N' semi-integer variable
    */
    if (probtype != CPXPROB_LP && probtype != CPXPROB_QP
            && probtype != CPXPROB_QCP)
    {

        ctype = (char *)malloc(ncols*sizeof(char));
        status = CPXgetctype (env, lp, ctype, 0, ncols-1);
        ctype[ncols]='\0';
        if (*disp == 1)
            printf("----------variable types----------:\n%s\n",ctype);
        VARTYPE_OUT = mxCreateString(ctype);

    }
    else
    {
        if (*disp == 1)
        {
            printf("-----variable types-----:continuous variables\n");
        }
        VARTYPE_OUT = mxCreateDoubleMatrix(0, 0, mxREAL);
    }

    // 10- Quadratic convex constraints (reserve)
    QC_OUT = mxCreateDoubleMatrix(0,0,mxREAL);

TERMINATE:

    return (status);

}  /* END readprob */

static void
free_and_null (char **ptr)
{
    if ( *ptr != NULL )
    {
        free (*ptr);
        *ptr = NULL;
    }
```

```
} /* END free_and_null */




static void
printVF        (double *v, int nvars)
{
    int i=0;
    for (i=0;i<nvars;i++)
    {
        printf("%f ",v[i]);
    }
    printf("\n");
} /* END printV */

static void
printVI        (int *v, int nvars)
{
    int i=0;
    for (i=0;i<nvars;i++)
    {
        printf("%d ",v[i]);
    }
    printf("\n");
} /* END printV */

static void
printMF        (double *matrix, int nrows, int ncols)
{
    int i,j;
    for (i=0;i<nrows;i++)
    {
        for (j=0;j<ncols;j++)
        {
            printf("%f ",matrix[j*nrows+i]);
        }
        printf("\n");
    }
}

static void
printMI        (int *matrix, int nrows, int ncols)
{
    int i,j;
```

```
    for (i=0;i<nrows;i++)
    {
        for (j=0;j<ncols;j++)
        {
            printf("%d ",matrix[j*nrows+i]);
        }
        printf("\n");
    }
}

static void
GetMatrix        (double *matrix, int *cmatbeg, int *cmatind,
           double *cmatval, int cmatsz, int nrows, int ncols)
{
    int *cmatend;
    int i,j,k;

    cmatend = (int *)malloc((cmatsz+1)*sizeof(int));
    for (j=0;j<ncols-1;j++)
    {
        cmatend[j] = cmatbeg[j+1]-1;
    }
    cmatend[ncols-1] = cmatsz-1;

    for (j=0;j<ncols;j++)
    {
        for (k=cmatbeg[j];k<=cmatend[j];k++)
        {
            i = cmatind[k];
            matrix[j*nrows+i] = cmatval[k];
        }
    }
}
```

Example : Using this function is easy, for example :
We can call the function in MATLAB as

```
[STAT,H,f,A,b,INDEQ,QC,LB,UB,VARTYPE] = CPXreadprob('mylp.mps',2);
```

This command is used for reading a problem file such as "mylp.mps" and "mypl.lp" to obtain the data from the MATLAB variables STAT,H,f,A,b,INDEQ,QC,LB,UB and VARTYPE. The meaning of these variables is presented as follows :

```
H          An (n x n) SYMETRIC, POSITIVE SEMIDEFINITE matrix (in full or
           sparse format) containing the quadratic objective function
```

```
           coefficients. Default: [], (no quadratic cost).

f          An (n x 1) vector containing the linear objective function
           coefficients. REQUIRED INPUT ARGUMENT.

A          An (m x n) matrix (in full or sparse format) containing the
           constraint coefficients. REQUIRED INPUT ARGUMENT.

b          An (m x 1) vector containing the right-hand side value for each
           constraint in the constraint matrix. REQUIRED INPUT ARGUMENT.

INDEQ      A vector containing the indices of equality constraints, i.e.,
              A(INDEQ,:) x = B(INDEQ,:).
           Default: [], (no equality constraints).

QC         A structure array containing Quadratic Constraints of the type
              x' * QC(i).Q * x + QC(i).L * x <= QC(i).r, i=1,...,nQC,
           where nQC is the total number of quadratic constraints.
  QC(i).Q     - An (n x n) POSITIVE SEMIDEFINITE matrix (in full or sparse
                format), quadratic part of the quadratic contraint i.
  QC(i).L     - A (1 x n) vector, linear part of the QC i.
  QC(i).r     - A scalar, right hand side of the quadratic constraint i.
           Default: [], (no quadratic constraints).

           NOTE: Contrary to the expressions for the objective function,
           in quadratic constraints we do not multiply quadratic term
           with 0.5. Also note that QC(i).L is a row vector.

LB         An (n x 1) vector containing the lower bound on each of the
           variables. Any lower bound that is set to a value less than
           or equal to that of the constant -CPX_INFBOUND will be
           treated as negative \infty. CPX_INFBOUND is defined in the
           header file cplex.h. Default: [], (lower bound of all
           variables set to -CPX_INFBOUND).

UB         An (n x 1) vector containing the upper bound on each of the
           variables. Any upper bound that is set to a value greater
           than or equal to that of the constant CPX_INFBOUND will be
           treated as \infty. CPX_INFBOUND is defined in the header
           file cplex.h. Default: [], (upper bound of all variables set
           to CPX_INFBOUND).

VARTYPE    An (n x 1) vector containing the types of the variables
           VARTYPE(i) = 'C' Continuous variable
           VARTYPE(i) = 'B' Binary(0/1) variable
```

```
VARTYPE(i) = 'I' Integer variable
VARTYPE(i) = 'S' Semi-continuous variable
VARTYPE(i) = 'N' Semi-integer variable
(This is case sensitive).
Default: [], (all variables are continuous).
```

A.2.2 Write MATLAB data to .mps and .lp file formats

Here is some parts of MATLAB mex code for writing problems :

```
/* ------------------------------------------------------------------------
 * file:       CPXwriteprob.c
 * Project:    MATLAB MEX interface for reading writing supported problems
 *             (as lp,mps etc)
 *
 * Purpose:    Invoke the CPLEX callable lib.
 *
 * Authors:     Yi-Shuai NIU
 * Contact:     INSA de Rouen
 *              niuys@insa-rouen.fr
 *
 * History: date: 2008.01.21  ver1.1
 * ------------------------------------------------------------------------
 *
 * Notes       This file requires CPLEX (9.0 or higher version)
 *             and MATLAB (6.0 or higher) to compile as CPXreadprob.mexw32
 *
 *              CPLEX    http://www.ilog.com
 *              MATLAB   http://www.mathworks.com
 *
 * (C) Jan. 21 2008 by Yi-Shuai NIU
 * Revision: 1.1
 * All rights reserved.
 */------------------------------------------------------------------------
/* Comments:
  Matlab interface for CPLEX 9.0 solver or higher for the following
  optimization problem

     min    0.5*x'*H*x + f'*x
      x
     s.t.:  A x {'<=' | '='} b
            x' * QC(i).Q * x + QC(i).L * x <= QC(i).r,  i=1,...,nQC
            LB <= x <= UB
            x(i) is {'C' = continuous | 'B' = binary | 'I' = integer |
            'S' = semi-continuous | 'N' = semi-integer}, i=1,...,n

  The calling syntax is:
  STAT = CPXwriteprob(NAME, H, f, A, b, INDEQ, QC, LB, UB,...
                                        VARTYPE, PARAM, OPTIONS)
*/

/* MATLAB declarations. */
```

```
#include <stdlib.h>
#include <matrix.h>
#include "mex.h"

/* CPLEX declarations.  */
#include "cplex.h"


#define VERSION "1.1"
#define COPYRIGHT "Copyright (C) 2008  Yi-Shuai NIU"

static CPXENVptr env = NULL;
static CPXFILEptr LogFile = NULL;

/* Problem type.
   We introduce our definition because getting the problem type
   from CPLEX env proved not to be the smartest idea.
   Maybe in the future CPLEX will handle this in a better way.
 */
enum{minLP, minQP, minMILP, minMIQP, minQCLP, minQCQP,
minQCMILP, minQCMIQP};


/* MEX Input Arguments */
enum {NAME_IN_POS, H_IN_POS, F_IN_POS, A_IN_POS, B_IN_POS, INDEQ_IN_POS,
QC_IN_POS, LB_IN_POS, UB_IN_POS, VARTYPE_IN_POS, PARAM_IN_POS,
OPTIONS_IN_POS,MAX_NUM_IN_ARG};

#define MIN_NUM_IN_ARG      5

#define NAME_IN      prhs[NAME_IN_POS]
#define H_IN         prhs[H_IN_POS]
#define F_IN         prhs[F_IN_POS]
#define A_IN         prhs[A_IN_POS]
#define B_IN         prhs[B_IN_POS]
#define INDEQ_IN     prhs[INDEQ_IN_POS]
#define QC_IN        prhs[QC_IN_POS]
#define LB_IN        prhs[LB_IN_POS]
#define UB_IN        prhs[UB_IN_POS]
#define VARTYPE_IN   prhs[VARTYPE_IN_POS]
#define PARAM_IN     prhs[PARAM_IN_POS]
#define OPTIONS_IN   prhs[OPTIONS_IN_POS]


/* MEX Output Arguments */
```

```
enum {STAT_OUT_POS,MAX_NUM_OUT_ARG};

/* If this is the only way to force people to check
   SOLSTAT before interpreting results then so be it! */
#define MIN_NUM_OUT_ARG     0
#define MAX_NUM_OUT_ARG     1

#define STAT_OUT     plhs[STAT_OUT_POS]


#define MAX_STR_LENGTH      1024


#if !defined(MAX)
#define MAX(A, B)   ((A) > (B) ? (A) : (B))
#endif

#if !defined(MIN)
#define MIN(A, B)   ((A) < (B) ? (A) : (B))
#endif

/* This hack is because Matlab R14 can crash on Linux due to the call to
   mexErrMsgTxt() */
#define TROUBLE_mexErrMsgTxt(A)    mexPrintf(A); mexPrintf("\n"); return
/* Here is the original version
#define TROUBLE_mexErrMsgTxt(A)    mexErrMsgTxt(A)
*/

/* this is for TRYING to release the CPLEX license when pressing CTRL-C */
#define RELEASE_CPLEX_LIC   1

/*
Copy and transform Matlab matrix IN in the representation needed for CPLEX.
Return number of non zero elements (nnz) and CPLEX matrix description:
matbeg, matcnt, matind, and matval.
The arrays matbeg, matcnt, matind, and matval are accessed as follows.
Suppose that CPLEX wants to access the entries in some column j. These
are assumed to be given by the array entries:
    matval[matbeg[j]],.., matval[matbeg[j]+matcnt[j]-1]
The corresponding row indices are:
    matind[matbeg[j]],.., matind[matbeg[j]+matcnt[j]-1]
Entries in matind are not required to be in row order. Duplicate entries
in matind within a single column are not allowed. The length of the arrays
matbeg and matind should be of at least numcols. The length of arrays
matind and matval should be of at least matbeg[numcols-1]+matcnt[numcols-1].
```

```
*/

int get_matrix(const mxArray *IN, int **outbeg, int **outcnt, int **outind,
               double **outval)
{
    int i, j;
    int gcount = 0;
    int pcount = 0;

    int m = mxGetM(IN);
    int n = mxGetN(IN);

    int    *matbeg = NULL;
    int    *matcnt = NULL;
    int    *matind = NULL;
    double *matval = NULL;

    double *in = mxGetPr(IN);


    matbeg = (int *)mxCalloc(n, sizeof(int));
    matcnt = (int *)mxCalloc(n, sizeof(int));

    /* Use different approaches for full and sparse matrix IN. */
    if (!mxIsSparse(IN))
    {

        gcount = 0;
        for (i = 0; i < n; i++)
        {
            pcount = 0;
            for (j = 0; j < m; j++)
            {
                if (in[i * m + j] != 0)
                {
                    gcount++;
                    pcount++;
                }
                matbeg[i] = gcount - pcount;
                matcnt[i] = pcount;
            }
        }
        matind = (int *)mxCalloc(gcount, sizeof(int));
        matval = (double *)mxCalloc(gcount, sizeof(double));
```

```
        gcount = 0;
        for (i = 0; i < n; i++)
        {
            for (j = 0; j < m; j++)
            {
                if (in[i * m + j] != 0)
                {
                    matind[gcount] = j;
                    matval[gcount] = in[i * m + j];
                    gcount++;
                }
            }
        }
    }
    else
    {
        /* For sparse matrix majority is already defined. */
        gcount = mxGetJc(IN)[n];
        matind = (int *)mxCalloc(gcount, sizeof(int));
        matval = (double *)mxCalloc(gcount, sizeof(double));

        for (i=0; i<n; i++)
        {
            matbeg[i] = mxGetJc(IN)[i];
            matcnt[i] = mxGetJc(IN)[i+1] - mxGetJc(IN)[i];
        }
        for (i=0; i<gcount; i++)
        {
            matind[i] = mxGetIr(IN)[i];
            matval[i] = in[i];
        }
    }
    *outbeg = matbeg;
    *outcnt = matcnt;
    *outind = matind;
    *outval = matval;
    return (gcount);
}

/*
Copy and transform Matlab matrix IN in the vector representation needed for
CPLEX. Return number of non zero elements (nnz) and CPLEX vector
description: matval.
*/
int get_vector(const mxArray *IN, double **outval)
```

```
{
    int i;
    int gcount = 0;

    int m = mxGetM(IN);
    int n = mxGetN(IN);

    double *matval = NULL;

    double *in = mxGetPr(IN);

    matval = (double *)mxCalloc(m*n, sizeof(double));

    gcount = 0;
    for (i = 0; i < m*n; i++)
    {
        matval[i] = in[i];
        /* Handle infinity entries */
        /*
        if (matval[i]==mxGetInf()){
            matval[i]=CPX_INFBOUND;
        } else if (matval[i]==-mxGetInf()) {
            matval[i]=-CPX_INFBOUND;
        }
        */
        if (in[i] != 0)
            gcount++;
    }
    *outval = matval;
    return (gcount);
}

/*
    Display CPLEX error code message
 */
void dispCPLEXerror(CPXENVptr env, int status)
{
    char errmsg[MAX_STR_LENGTH];
    char *errstr;

    errstr = (char *)CPXgeterrorstring (env, status, errmsg);
    if ( errstr != NULL )
    {
        mexPrintf("%s",errmsg);
    }
```

```
    else
    {
        mexPrintf("CPLEX Error %5d:  Unknown error code.\n", status);
    }
}


/*
   Here is the exit function, which gets run when the MEX-file is
   cleared and when the user exits MATLAB. The mexAtExit function
   should always be declared as static.
 */
static void freelicence(void)
{
    int             status;
    extern CPXENVptr env;
    extern CPXFILEptr LogFile;

    /* Close log file */
    if (LogFile != NULL)
    {
        mexPrintf("LogFile is not NULL.\n");
        status=CPXfclose(LogFile);

        if (status)
        {
            mexPrintf("Could not close log file cplex_logfile.log.\n");
        }
        else
        {
        /* Just to be on the safe side we declare that the LogFile after
        closing is NULL. In this way we avoid possible error when trying
        to clear the same mex file more than once. */
            LogFile = NULL;
        }
    }
    else
    {
        /* mexPrintf("LogFile is NULL.\n"); */
    }


    /* Close CPLEX environment */
    if (env != NULL)
    {
```

```
        /* mexPrintf("env is not NULL.\n"); */
        status = CPXcloseCPLEX(&env);
        /*
        Note that CPXcloseCPLEX produces no output,
        so the only way to see the cause of the error is to use
        CPXgeterrorstring.  For other CPLEX routines, the errors will
        be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.
        */
        if (status)
        {
            mexPrintf("Could not close CPLEX environment.\n");
            dispCPLEXerror(env, status);
        }
        else
        {
        /* Just to be on the safe side we declare that the environment after
         closing is NULL. In this way we avoid possible error when trying
         to clear the same mex file more than once. */
            env = NULL;
        }
    }
    else
    {
        /* mexPrintf("env is NULL.\n"); */
    }

}

/**********************************
 *                                *
 *          MATLAB interface      *
 *                                *
 **********************************/
void mexFunction(int nlhs, mxArray * plhs[], int nrhs,
const mxArray * prhs[])
{
    int         probtype = minLP;  /* default optimization problem type */

    char        errmsg[MAX_STR_LENGTH];     /* buffer for error messages */

    /* tmp variables */
    int             i, ii, jj, kk, mm;
    char           *tmpc;
    double         *tmpd;
    mxArray        *tmpArr;
```

```
int             n_vars = 0;
int             n_constr = 0;

/* A_IN variables */
int             A_nnz = 0;          /* number of non-zero elements */
int            *A_matbeg = NULL;
int            *A_matcnt = NULL;
int            *A_matind = NULL;
double         *A_matval = NULL;

/* B_IN variables */
int             b_nnz = 0;          /* number of non-zero elements */
double         *b_matval = NULL;

/* H_IN variables */
int             H_nnz = 0;          /* number of non-zero elements */
int            *H_matbeg = NULL;
int            *H_matcnt = NULL;
int            *H_matind = NULL;
double         *H_matval = NULL;

/* F_IN variables */
int             f_nnz = 0;          /* number of non-zero elements */
double         *f_matval = NULL;

/* INDEQ_IN variables */
int             sense_nnz = 0;      /* number of non-zero elements */
char           *sense = NULL;       /* is constraint 'L' or 'E'    */

/* Quadratic Constraint QC_IN variables */
int             nQC = 0;            /* number of the QC */
double         *QC_r = NULL;        /* right hand side of the QC */
int            *QC_linnzcnt = NULL; /* linear part of the QC,
                                       number of nonzero elements */
int           **QC_linind = NULL;   /* linear part of the QC,
                                       indices of nonzero elements */
double        **QC_linval = NULL;   /* linear part of the QC,
                                       values of nonzero elements */
int            *QC_quadnzcnt = NULL;/* quadratic part of the QC,
                                       number of nonzero elements */
int           **QC_quadrow = NULL;  /* quadratic part of the QC,
                                      row indices of nonzero elements */
int           **QC_quadcol = NULL;  /* quadratic part of the QC,
                                      column indices of nnz elements */
```

```
double        **QC_quadval = NULL;  /* quadratic part of the QC,
                                       values of nonzero elements */


/* LB_IN variables */
int             LB_nnz = 0;          /* number of non-zero elements */
double         *LB_matval = NULL;

/* UB_IN variables */
int             UB_nnz = 0;          /* number of non-zero elements */
double         *UB_matval = NULL;

/* VARTYPE_IN variables */
int             vartype_nnC = 0; /* number of non-continuous elements */
char           *vartype = NULL;

/* PARAM_IN variables */
int             nintpar = 0;
double         *intparcode = NULL;
double         *intparvalue = NULL;
int             ndoublepar = 0;
double         *doubleparcode = NULL;
double         *doubleparvalue = NULL;


/* OPTIONS_IN variables */
const char     *opt_fnames[] =
{"verbose","save_prob","x0","probtype","lic_rel"};
int             opt_nfields = (sizeof(opt_fnames)/sizeof(*opt_fnames));
int             opt_verbose = 0;
int             opt_save_probind = 0;
char           *opt_save_prob = NULL;
int             opt_logfile = 0;
int             opt_nx0 = 0;
int            *opt_x0i = NULL;
double         *opt_x0 = NULL;
int             opt_probtype = -1;
/* user can specify problem type, -1 internaly means: no specification
                                       by the user */
int             opt_lic_rel = 1;
/* user can specify after how many calls will
CPLEX environment be closed and license released */


/* STAT_OUT variables */
```

```
mxArray        *STAT = NULL;
double         *stat = NULL;



/* CPLEX variables */
char *probname = NULL;
extern CPXENVptr env;
extern CPXFILEptr LogFile;

CPXLPptr        lp = NULL;
int             status;
int             cplex_probtype;

int             errors = 1;     /* indicator of success */



/* If there are no input nor output arguments display version number */
if ((nrhs == 0) && (nlhs == 0))
{
    mexPrintf("Version %s.\n", VERSION);
    mexPrintf("MEX interface for writing CPLEX file in Matlab.\n");
    mexPrintf("%s.\n", COPYRIGHT);
    return;
}

/* Check for proper number of arguments. */
if (nrhs < MIN_NUM_IN_ARG)
{
    sprintf(errmsg, "At least %d input arguments required.",
            MIN_NUM_IN_ARG);
    TROUBLE_mexErrMsgTxt(errmsg);
}
else if (nrhs > MAX_NUM_IN_ARG)
{
    TROUBLE_mexErrMsgTxt("Too many input arguments.");
}
else if (nlhs < MIN_NUM_OUT_ARG)
{
mexPrintf("NOTE: notice SOLSTAT output.\n");
mexPrintf("check SOLSTAT for correct interpretation of the results.\n");
sprintf(errmsg, "At least %d output arguments required.",
            MIN_NUM_OUT_ARG);
    TROUBLE_mexErrMsgTxt(errmsg);
```

```
}
else if (nlhs > MAX_NUM_OUT_ARG)
{
    TROUBLE_mexErrMsgTxt("Too many output arguments.");
}

/* Is somebody trying to solve an unconstrained problem. */
if ((mxIsEmpty(A_IN)) || (mxIsEmpty(B_IN)))
{
mexPrintf("If you are trying to solve an unconstrained problem\n");
mexPrintf("artificially introduce BIG constraints never be active.\n");
TROUBLE_mexErrMsgTxt("CPLEX requires non-empty matrices A and b.");
}


mxGetPr(NAME_IN);
i = mxGetNumberOfElements(NAME_IN)+1;
/* Allocate enough memory to hold the converted string. */
probname = mxCalloc(i, sizeof (char));

/* Copy the string data */
if (mxGetString(NAME_IN, probname, i) != 0)
{
    TROUBLE_mexErrMsgTxt("Could not convert string data.");
}

/* First get the number of variables and the number of constraints.
   For this purpose check size of a matrix A_IN (required argument). */
if (   (!mxIsNumeric(A_IN))
        || (mxGetNumberOfDimensions(A_IN) > 2)
        || ((n_constr = mxGetM(A_IN)) < 1)
        || ((n_vars = mxGetN(A_IN)) < 1)
        || (mxIsComplex(A_IN))
        || (mxGetPr(A_IN) == NULL)
   )
{
    TROUBLE_mexErrMsgTxt("Matrix A must be a real valued (m x n)
                         matrix, with m>=1, n>=1.");
}
A_nnz = get_matrix(A_IN, &A_matbeg, &A_matcnt, &A_matind, &A_matval);
if (A_nnz == 0)
{
    TROUBLE_mexErrMsgTxt("At least one element of constraint matrix A
                         must be non-zero.");
}
```

```
if (   (!mxIsNumeric(B_IN))
        || (mxGetNumberOfDimensions(B_IN) > 2)
        || (mxGetM(B_IN) != n_constr)
        || (mxGetN(B_IN) != 1)
        || (mxIsComplex(B_IN))
        || (mxGetPr(B_IN) == NULL)
   )
{
    sprintf(errmsg,
    "Vector b must be a real valued (%d x 1) vector.", n_constr);
    TROUBLE_mexErrMsgTxt(errmsg);
}
b_nnz = get_vector(B_IN, &b_matval);


if (!mxIsEmpty(H_IN))
{
    if (!mxIsNumeric(H_IN)
            || (mxGetNumberOfDimensions(H_IN) > 2)
            || (mxGetM(H_IN) != n_vars)
            || (mxGetN(H_IN) != n_vars)
            || (mxIsComplex(H_IN))
            || (mxGetPr(H_IN) == NULL)
       )
    {
        sprintf(errmsg,
        "If non-empty, H must be a real valued (%d x %d) matrix.",
        n_vars, n_vars);
        TROUBLE_mexErrMsgTxt(errmsg);
    }
    H_nnz = get_matrix(H_IN, &H_matbeg, &H_matcnt,
    &H_matind, &H_matval);
}


/* Is somebody trying to solve a feasibility problem. */
if ((mxIsEmpty(F_IN)) && (H_nnz == 0))
{
TROUBLE_mexErrMsgTxt("CPLEX requires non-empty objective vector f.");
}

if (   (!mxIsNumeric(F_IN))
        || (mxGetNumberOfDimensions(F_IN) > 2)
```

```
        || (mxGetM(F_IN) != n_vars)
        || (mxGetN(F_IN) != 1)
        || (mxIsComplex(F_IN))
        || (mxGetPr(F_IN) == NULL)
   )
{
    sprintf(errmsg,
            "Objective f must be a real valued (%d x 1) vector.",
            n_vars);
    TROUBLE_mexErrMsgTxt(errmsg);
}
f_nnz = get_vector(F_IN, &f_matval);


/* Initially we assume that all constraints are of '<=' type */
sense = mxCalloc(n_constr+1, sizeof(char));
for (i = 0; i < n_constr; i++)
    sense[i] = 'L';
sense[n_constr] = 0;

/* Now check if there are some equality constraints */
if ((nrhs > INDEQ_IN_POS) && (!mxIsEmpty(INDEQ_IN)))
{
    if (   (!mxIsNumeric(INDEQ_IN))
            || (mxGetNumberOfDimensions(INDEQ_IN) > 2)
            || (MIN((mxGetM(INDEQ_IN)), (mxGetN(INDEQ_IN))) != 1)
            || (mxIsComplex(INDEQ_IN))
       )
    {
    TROUBLE_mexErrMsgTxt("INDEQ must be a vector of indices of
    equality constraints.");
    }
    else
    {
        tmpd = mxGetPr(INDEQ_IN);
        for (i = 0; i < mxGetNumberOfElements(INDEQ_IN); i++)
        {
            if ((tmpd[i]>=1) && (tmpd[i]<=n_constr))
            {
                sense[(int)tmpd[i]-1] = 'E';
            }
            else
            {
                TROUBLE_mexErrMsgTxt("Index in INDEQ points to a
                non-existing constraint.");
```

```
            }
        }
    }
}
sense_nnz = n_constr;


if ((nrhs > QC_IN_POS) && (mxIsStruct(QC_IN)))
{

    if ((mxGetNumberOfDimensions(QC_IN) > 2)
            || (MIN(mxGetM(QC_IN),mxGetN(QC_IN)) > 1))
    {
        TROUBLE_mexErrMsgTxt("QC must be a STRUCTURE array.");
    }
    else
    {
        nQC = mxGetM(QC_IN)*mxGetN(QC_IN);
    }

    QC_r = (double *)mxCalloc(nQC, sizeof(double));
    QC_linnzcnt = (int *)mxCalloc(nQC, sizeof(int));
    QC_linind = (int **)mxCalloc(nQC, sizeof(int *));
    QC_linval = (double **)mxCalloc(nQC, sizeof(double *));
    QC_quadnzcnt = (int *)mxCalloc(nQC, sizeof(int));
    QC_quadrow = (int **)mxCalloc(nQC, sizeof(int *));
    QC_quadcol = (int **)mxCalloc(nQC, sizeof(int *));
    QC_quadval = (double **)mxCalloc(nQC, sizeof(double *));

    for (ii = 0; ii < nQC; ii++)
    {
        QC_quadnzcnt[ii] = 0;
        if (((tmpArr = mxGetField(QC_IN, ii, "Q"))  !=NULL)  &&
                (mxGetNumberOfDimensions(tmpArr) == 2) &&
                (mxGetM(tmpArr)==n_vars) &&
                (mxGetN(tmpArr)==n_vars) &&
                ((tmpd = mxGetPr(tmpArr)) !=NULL))
        {
            if (!mxIsSparse(tmpArr))
            {
                for (jj=0; jj<n_vars; jj++)
                {
                    for (kk=0; kk<n_vars; kk++)
                    {
                        /* count nonzero entries */
```

```
                    if (tmpd[jj*n_vars+kk] !=0)
                    {
                        QC_quadnzcnt[ii] = QC_quadnzcnt[ii]+1;
                    }
                }
            }
        }
        else
        {
            QC_quadnzcnt[ii] = mxGetJc(tmpArr)[n_vars];
        }
    }
    else
    {
        sprintf(errmsg,
                "QC(%d) must have a %d by %d matrix Q as a field.",
                ii+1,n_vars, n_vars);
        TROUBLE_mexErrMsgTxt(errmsg);
    }

/* Now that we know the number of nonzero entries for this QC. */
/* Let's allocate the space for it and fill it in.             */
QC_quadrow[ii] = (int *)mxCalloc(QC_quadnzcnt[ii], sizeof(int));
QC_quadcol[ii] = (int *)mxCalloc(QC_quadnzcnt[ii], sizeof(int));
QC_quadval[ii] = (double *)mxCalloc(QC_quadnzcnt[ii],
sizeof(double));

    if (!mxIsSparse(tmpArr))
    {
        mm = 0;
        for (jj=0; jj<n_vars; jj++)
        {
            for (kk=0; kk<n_vars; kk++)
            {
                /* MATLAB stores matrices columnwise !!! */
                if (tmpd[jj*n_vars+kk] !=0)
                {
                    QC_quadrow[ii][mm] = kk;
                    QC_quadcol[ii][mm] = jj;
                    QC_quadval[ii][mm] = tmpd[jj*n_vars+kk];
                    mm++;
                }
            }
        }
    }
```

```
            else
            {
                mm = 0;
                for (jj=0; jj<n_vars; jj++)
                {
                for (kk=0; kk < mxGetJc(tmpArr)[jj+1]-mxGetJc(tmpArr)[jj];
                kk++)
                    {
                        QC_quadrow[ii][mm] = mxGetIr(tmpArr)[mm];
                        QC_quadcol[ii][mm] = jj;
                        QC_quadval[ii][mm] = tmpd[mm];
                        mm++;
                    }
                }
            }


            QC_linnzcnt[ii] = 0;
            if (((tmpArr = mxGetField(QC_IN, ii, "L"))  !=NULL)  &&
                    (mxGetNumberOfDimensions(tmpArr) == 2) &&
                    (mxGetM(tmpArr)==1) &&
                    (mxGetN(tmpArr)==n_vars) &&
                    ((tmpd = mxGetPr(tmpArr)) !=NULL) )
            {
                for (jj=0; jj<n_vars; jj++)
                {
                    /* count nonzero entries */
                    if (tmpd[jj] !=0)
                    {
                        QC_linnzcnt[ii] = QC_linnzcnt[ii] + 1;
                    }
                }
            }
            else
            {
                sprintf(errmsg,
                        "QC(%d) must have a 1 by %d vector L as a field.",
                        ii+1, n_vars);
                TROUBLE_mexErrMsgTxt(errmsg);
            }

        /* Now that we know the number of nonzero entries for this QC. */
        /* Let's allocate the space for it and fill it in.             */
        QC_linind[ii] = (int *)mxCalloc(QC_linnzcnt[ii], sizeof (int));
        QC_linval[ii] = (double *)mxCalloc(QC_linnzcnt[ii], sizeof (double));
```

```
            mm = 0;
            for (jj=0; jj<n_vars; jj++)
            {
                if (tmpd[jj] !=0)
                {
                    QC_linind[ii][mm] = jj;
                    QC_linval[ii][mm] = tmpd[jj];
                    mm++;
                }
            }


            if (((tmpArr = mxGetField(QC_IN, ii, "r"))  !=NULL)  &&
                    (mxGetNumberOfDimensions(tmpArr) == 2) &&
                    (mxGetM(tmpArr)==1) &&
                    (mxGetN(tmpArr)==1) &&
                    ((tmpd = mxGetPr(tmpArr)) !=NULL) )
            {
                QC_r[ii]=tmpd[0];
            }
            else
            {
                sprintf(errmsg,
                        "QC(%d) must have a 1 by 1 scalar r as a field.",
                        ii+1);
                TROUBLE_mexErrMsgTxt(errmsg);
            }
        } /* ii<nQC */
    } /* QC_IN_POS */



    if ((nrhs > LB_IN_POS) && (!mxIsEmpty(LB_IN)))
    {
        if (   (!mxIsNumeric(LB_IN))
                || (mxGetNumberOfDimensions(LB_IN) > 2)
                || (mxGetM(LB_IN) != n_vars)
                || (mxGetN(LB_IN) != 1)
                || (mxIsComplex(LB_IN))
                || (mxGetPr(LB_IN) == NULL)
           )
        {
            sprintf(errmsg,
                    "LB must be a real valued (%d x 1) column vector.",
                    n_vars);
```

```
            TROUBLE_mexErrMsgTxt(errmsg);
        }
        LB_nnz = get_vector(LB_IN, &LB_matval);
    }
    if (LB_matval == NULL)
    {
        LB_matval = mxCalloc(n_vars, sizeof(double));
        for (i=0; i<n_vars; i++)
        {
            LB_matval[i] = -mxGetInf();
        }
        LB_nnz = n_vars;
    }


    if ((nrhs > UB_IN_POS) && (!mxIsEmpty(UB_IN)))
    {
        if (   (!mxIsNumeric(UB_IN))
                || (mxGetNumberOfDimensions(UB_IN) > 2)
                || (mxGetM(UB_IN) != n_vars)
                || (mxGetN(UB_IN) != 1)
                || (mxIsComplex(UB_IN))
                || (mxGetPr(UB_IN) == NULL)
           )
        {
            sprintf(errmsg,
                    "UB must be a real valued (%d x 1) column vector.",
                    n_vars);
            TROUBLE_mexErrMsgTxt(errmsg);
        }
        UB_nnz = get_vector(UB_IN, &UB_matval);
    }
    if (UB_matval == NULL)
    {
        UB_matval = mxCalloc(n_vars, sizeof(double));
        for (i=0; i<n_vars; i++)
        {
            UB_matval[i] = mxGetInf();
        }
        UB_nnz = n_vars;
    }


    if ((nrhs > VARTYPE_IN_POS) && (!mxIsEmpty(VARTYPE_IN)))
    {
```

```
        if (   (!mxIsChar(VARTYPE_IN))
                || (mxGetNumberOfDimensions(VARTYPE_IN) > 2)
                || (mxGetM(VARTYPE_IN) != n_vars)
                || (mxGetN(VARTYPE_IN) != 1)
           )
        {
            sprintf(errmsg,
            "VARTYPE must be a char valued (%d x 1) column vector.",
            n_vars);
            TROUBLE_mexErrMsgTxt(errmsg);
        }
        else
        {
            /* Allocate enough memory to hold the converted string. */
            vartype = mxCalloc(n_vars+1, sizeof (char));

            /* Copy the string data from string_array_ptr into buf. */
            if (mxGetString(VARTYPE_IN, vartype, n_vars+1) != 0)
            {
            TROUBLE_mexErrMsgTxt("Could not convert VARTYPE.");
            }


            /* Checking if the input is made only of C B I S N. */
            vartype_nnC = 0;
            for (i = 0; i < n_vars; i++)
            {
                if (vartype[i] != 'C')
                {
                    vartype_nnC++;
                }
                if (   (vartype[i] != 'C')
                        && (vartype[i] != 'B')
                        && (vartype[i] != 'I')
                        && (vartype[i] != 'S')
                        && (vartype[i] != 'N'))
                {
                TROUBLE_mexErrMsgTxt("VARTYPE must contain only C,B,I,S,N");
                }
            }
        }
    }
    if (vartype == NULL)
    {
        vartype = mxCalloc(n_vars + 1, sizeof(char));
```

```
        for (i = 0; i < n_vars; i++)
            vartype[i] = 'C';
        vartype[n_vars] = 0;
        vartype_nnC = 0;
    }


    if ((nrhs > PARAM_IN_POS) && (mxIsStruct(PARAM_IN)))
    {
        if (   ((tmpArr = mxGetField(PARAM_IN, 0, "int"))  !=NULL)
                && (mxGetNumberOfDimensions(tmpArr) == 2)
                && (mxGetM(tmpArr)>=1)
           )
        {
            if (mxGetN(tmpArr)==2)
            {
                nintpar=mxGetM(tmpArr);

                intparcode = mxCalloc(nintpar, sizeof(double));
                intparvalue = mxCalloc(nintpar, sizeof(double));

                tmpd=mxGetPr(tmpArr);
                for (i=0; i<nintpar; i++)
                {
                    intparcode[i]=tmpd[i];
                    intparvalue[i]=tmpd[i+nintpar];
                }
            }
            else
            {
                TROUBLE_mexErrMsgTxt("PARAM.int must be a matrix of
                size nintpar x 2.");
            }
        }
        if (   ((tmpArr = mxGetField(PARAM_IN, 0, "double")) !=NULL)
                && (mxGetNumberOfDimensions(tmpArr) == 2)
                && (mxGetM(tmpArr)>=1)
           )
        {
            if (mxGetN(tmpArr)==2)
            {
                ndoublepar=mxGetM(tmpArr);

                doubleparcode = mxCalloc(ndoublepar, sizeof(double));
                doubleparvalue = mxCalloc(ndoublepar, sizeof(double));
```

```
            tmpd=mxGetPr(tmpArr);
            for (i=0; i<ndoublepar; i++)
            {
                doubleparcode[i]=tmpd[i];
                doubleparvalue[i]=tmpd[i+ndoublepar];
            }
        }
        else
        {
            TROUBLE_mexErrMsgTxt("PARAM.double must be a matrix of
            size ndoublepar x 2.");
        }
    }
}


if (   (nrhs > OPTIONS_IN_POS) && (mxIsStruct(OPTIONS_IN)) )
{

    /* OPTIONS.verbose */
    if (   ((tmpArr = mxGetField(OPTIONS_IN, 0, "verbose"))  !=NULL)
            && (!mxIsEmpty(tmpArr))
       )
    {
        if (   (mxIsNumeric(tmpArr))
                && (mxGetNumberOfDimensions(tmpArr) == 2)
                && (mxGetM(tmpArr)==1)
                && (mxGetN(tmpArr)==1)
                && (!mxIsComplex(tmpArr))
           )
        {
            tmpd = mxGetPr(tmpArr);
            if ((tmpd[0] != 0) && (tmpd[0] != 1) && (tmpd[0] != 2))
            {
            TROUBLE_mexErrMsgTxt("OPTIONS.verbose must be 0, 1 or 2.");
            }
            opt_verbose = (int)tmpd[0];
        }
        else
        {
            TROUBLE_mexErrMsgTxt("OPTIONS.verbose must be 0, 1 or 2.");
        }
    }
```

```
/* OPTIONS.save_prob */
if (   ((tmpArr = mxGetField(OPTIONS_IN, 0, "save_prob"))  !=NULL)
        && (!mxIsEmpty(tmpArr))
   )
{
    if (   (mxIsChar(tmpArr))
            && (mxGetNumberOfDimensions(tmpArr) == 2)
       )
    {
        opt_save_probind = 1;
        i = mxGetNumberOfElements(tmpArr)+1;
        /* Allocate enough memory to hold the converted string. */
        opt_save_prob = mxCalloc(i, sizeof (char));

        /* Copy the string data */
        if (mxGetString(tmpArr, opt_save_prob, i) != 0)
        {
            TROUBLE_mexErrMsgTxt("Could not convert string data.");
        }
    }
    else
    {
        TROUBLE_mexErrMsgTxt("OPTIONS.save_prob must be a string.");
    }
}

/* OPTIONS.logfile */
if (   ((tmpArr = mxGetField(OPTIONS_IN, 0, "logfile"))  !=NULL)
        && (!mxIsEmpty(tmpArr))
   )
{
    if (   (mxIsNumeric(tmpArr))
            && (mxGetNumberOfDimensions(tmpArr) == 2)
            && (mxGetM(tmpArr)==1)
            && (mxGetN(tmpArr)==1)
            && (!mxIsComplex(tmpArr))
       )
    {
        tmpd = mxGetPr(tmpArr);
        if (   (tmpd[0] != 0)
                && (tmpd[0] != 1)
           )
        {
            TROUBLE_mexErrMsgTxt("OPTIONS.logfile must be 0 or 1.");
```

```
            }
            opt_logfile = (int)tmpd[0];
        }
        else
        {
            TROUBLE_mexErrMsgTxt("OPTIONS.logfile must be 0 or 1.");
        }
    }


    /* OPTIONS.x0 */
    if (   ((tmpArr = mxGetField(OPTIONS_IN, 0, "x0"))  !=NULL)
            && (!mxIsEmpty(tmpArr))
            && (mxGetNumberOfDimensions(tmpArr) == 2)
            && (!mxIsComplex(tmpArr))
       )
    {
        if (mxGetN(tmpArr)==2)
        {
            opt_nx0=mxGetM(tmpArr);

            opt_x0i = (int *)mxCalloc(opt_nx0, sizeof(int));
            opt_x0 = mxCalloc(opt_nx0, sizeof(double));

            tmpd=mxGetPr(tmpArr);
            for (i=0; i<opt_nx0; i++)
            {
                if (tmpd[i]<1 || tmpd[i]>n_vars)
                {
                    TROUBLE_mexErrMsgTxt("OPTIONS.x0 is indexing
                    non-existing variable.");
                }
                opt_x0i[i]=(int)tmpd[i] - 1;
                opt_x0[i]=tmpd[i+opt_nx0];
            }
        }
        else
        {
            TROUBLE_mexErrMsgTxt("OPTIONS.x0 must be a matrix of size
            nx0 x 2.");
        }
    }

    /* OPTIONS.probtype */
    if (   ((tmpArr = mxGetField(OPTIONS_IN, 0, "probtype"))  !=NULL)
```

```
            && (!mxIsEmpty(tmpArr))
       )
    {
        if (   (mxIsNumeric(tmpArr))
                && (mxGetNumberOfDimensions(tmpArr) == 2)
                && (mxGetM(tmpArr)==1)
                && (mxGetN(tmpArr)==1)
                && (!mxIsComplex(tmpArr))
           )
        {
            tmpd = mxGetPr(tmpArr);
            if (   (tmpd[0] != -1)
                    && (tmpd[0] != minLP)
                    && (tmpd[0] != minQP)
                    && (tmpd[0] != minMILP)
                    && (tmpd[0] != minMIQP)
                    && (tmpd[0] != minQCLP)
                    && (tmpd[0] != minQCQP)
                    && (tmpd[0] != minQCMILP)
                    && (tmpd[0] != minQCMIQP)
               )
            {
                TROUBLE_mexErrMsgTxt("Unknown problem type specified
                in OPTIONS.probtype.");
            }
            opt_probtype = (int)tmpd[0];
        }
        else
        {
            TROUBLE_mexErrMsgTxt("Unknown problem type specified in
            OPTIONS.probtype.");
        }
    }

    /* OPTIONS.lic_rel */
    if (   ((tmpArr = mxGetField(OPTIONS_IN, 0, "lic_rel"))  !=NULL)
            && (!mxIsEmpty(tmpArr))
       )
    {
        if (   (mxIsNumeric(tmpArr))
                && (mxGetNumberOfDimensions(tmpArr) == 2)
                && (mxGetM(tmpArr)==1)
                && (mxGetN(tmpArr)==1)
                && (!mxIsComplex(tmpArr))
           )
```

```
            {
                tmpd = mxGetPr(tmpArr);
                if (   (tmpd[0] < 1)
                        || (tmpd[0] != (int)tmpd[0])
                   )
                {
                    TROUBLE_mexErrMsgTxt("Wrong number of calls to CPLEX
                    before license is released see OPTIONS.lic_rel.");
                }
                opt_lic_rel = (int)tmpd[0];
            }
            else
            {
                TROUBLE_mexErrMsgTxt("Unknown problem type specified
                in OPTIONS.probtype.");
            }
        }

    } /* OPTIONS_IN */



    /*
       Register safe exit.
     */
#ifdef RELEASE_CPLEX_LIC
    mexAtExit(freelicence);
#endif


    /* Initialize the CPLEX environment. */
    env = CPXopenCPLEX(&status);


    /*
       If an error occurs, the status value indicates the reason for
       failure.  A call to CPXgeterrorstring will produce the text of
       the error message.  Note that CPXopenCPLEXdevelop produces no
       output, so the only way to see the cause of the error is to use
       CPXgeterrorstring.  For other CPLEX routines, the errors will
       be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.
     */
    if (env == NULL)
    {
        mexPrintf("Could not open CPLEX environment.\n");
```

```
        dispCPLEXerror(env, status);
        goto TERMINATE;
    }



    /* Create a log file. */
    if (opt_logfile)
    {
        /*
           Open a LogFile to print out any CPLEX messages in there.
           We do this since Matlab does not execute printf commands
           in MEX files properly under Windows.
         */
        LogFile = CPXfopen("cplex_logfile.log", "w");
        if (LogFile == NULL)
        {
        TROUBLE_mexErrMsgTxt("Could not open the log file .\n");
        }
        status = CPXsetlogfile(env, LogFile);
        if (status)
        {
            dispCPLEXerror(env, status);
            goto TERMINATE;
        }
    }

    /* Turn on output to the screen only if opt_verbose>=1. */
    if (opt_verbose>=1)
    {
        status = CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_OFF);
        if (status)
        {
            dispCPLEXerror(env, status);
            goto TERMINATE;
        }
    }

    /* Create the problem. */
    lp = CPXcreateprob(env, &status, probname);

    /*
       A returned pointer of NULL may mean that not enough memory
       was available or there was some other problem.  In the case of
       failure, an error message will have been written to the error
```

```
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.
 */
if (lp == NULL)
{
    mexPrintf("Failed to create LP.\n");
    dispCPLEXerror(env, status);
    goto TERMINATE;
}


/* Now copy the problem data into the lp. */
status = CPXcopylp(env, lp, n_vars, n_constr, CPX_MIN, f_matval,
b_matval, sense, A_matbeg, A_matcnt, A_matind, A_matval,LB_matval,
UB_matval, NULL);

if (status)
{
    mexPrintf("Failed to copy problem data.\n");
    dispCPLEXerror(env, status);
    goto TERMINATE;
}

/* Copy quadratic objective if one exists. */
if (H_nnz > 0)
{
    status = CPXcopyquad (env,lp,H_matbeg,H_matcnt,H_matind,H_matval);
    if (status)
    {
        mexPrintf("Failed to copy quadratic objective matrix H.\n");
        dispCPLEXerror(env, status);
        goto TERMINATE;
    }
}

/* Now copy the vartype array if it has non-continuous components */
if (vartype_nnC > 0)
{
    status = CPXcopyctype(env, lp, vartype);
    if (status)
    {
        mexPrintf("Failed to copy VARTYPE.\n");
        dispCPLEXerror(env, status);
        goto TERMINATE;
```

```
        }
    }

    /* Set initial solution. */
    if ((opt_nx0 > 0) && (vartype_nnC > 0))
    {
        status = CPXcopymipstart(env, lp, opt_nx0, opt_x0i, opt_x0);
        if (status)
        {
            mexPrintf("Failed to set initial solution.\n");
            dispCPLEXerror(env, status);
            goto TERMINATE;
        }
        /* modified by M. Kvasnica. according to CPLEX10 manual,
         * CPX_PARAM_ADVIND must be set to 1
         */
        status = CPXsetintparam(env, CPX_PARAM_ADVIND, 1);
    }


    /* Add quadratic constraints to the problem. */
    for (i = 0; i<nQC; i++)
    {
        status = CPXaddqconstr (env, lp, QC_linnzcnt[i], QC_quadnzcnt[i],
                                QC_r[i], 'L', QC_linind[i], QC_linval[i],
                                QC_quadrow[i], QC_quadcol[i], QC_quadval[i],
                                NULL);
        if (status)
        {
            mexPrintf("Failed to copy quadratic constraint.\n");
            dispCPLEXerror(env, status);
            goto TERMINATE;
        }
    }


    /* If verbose>=2 display problem on the screen. */
    if (opt_verbose>=2)
    {
        FILE *fp;
        char buf[80];

        mexPrintf("\nVERBOSITY IS ON!\n");

        /* Save problem to the verbosity file for display on the screen. */
```

```
        status = CPXwriteprob(env, lp, "cplex_verbose.lp", NULL);
        if (status)
        {
            mexPrintf("Failed to save the problem to the verbosity file.\n");
            dispCPLEXerror(env, status);
            goto TERMINATE;
        }
        mexPrintf("\nThe problem is stored in the file cplex_verbose.lp\n");
        if ((fp = fopen("cplex_verbose.lp", "r")) == NULL)
        {
            mexPrintf("Failed to open verbosity file.");
            goto TERMINATE;
        }

        while (fgets(buf, sizeof(buf), fp) != NULL)
            mexPrintf("%s", buf);
        fclose(fp);

        /* Initial guess */
        if ((opt_nx0) && (vartype_nnC > 0))
        {
            mexPrintf("\nStarting from:\n");
            for (i = 0; i < opt_nx0; i++)
            {
                mexPrintf("x%d = %15.10e\n", opt_x0i[i]+1, opt_x0[i]);
            }
        }
    }


    /* Save problem to the file. */
    if (opt_save_probind)
    {
        status = CPXwriteprob(env, lp, opt_save_prob, NULL);
        if (status)
        {
            mexPrintf("Failed to save problem to the file.\n");
            dispCPLEXerror(env, status);
            goto TERMINATE;
        }
    }



    /*
```

```
       Create matrices for the three main return arguments.
     */

    STAT = mxCreateDoubleMatrix(1, 1, mxREAL);
    stat = mxGetPr(STAT);

    /* SAVE problem to file with name probname */
    status = CPXwriteprob(env, lp, probname, NULL);
    if (status)
    {
        *stat=0;
        mexPrintf("Failed to save problem to the file.\n");
        dispCPLEXerror(env, status);
        goto TERMINATE;
    }
    else
    {
        *stat=1;
    }


TERMINATE:

    /* Close log file */
    if ((opt_logfile) && (LogFile != NULL))
    {
        status = CPXfclose(LogFile);

        if (status)
        {
            mexPrintf("Could not close log file cplex_logfile.log.\n");
        }
        else
        {
        /* Just to be on the safe side we declare that the LogFile after
        closing is NULL. In this way we avoid possible error when trying
        to clear the same mex file afterwards. */
            LogFile = NULL;
        }
    }


    /*
       Free up the problem as allocated by CPXcreateprob, if necessary.
     */
```

```
if (lp != NULL)
{
    status = CPXfreeprob(env, &lp);
    if (status)
    {
        if (opt_verbose>=1)
        {
            mexPrintf("CPXfreeprob failed.\n");
            dispCPLEXerror(env, status);
        }
    }
}

/* Free up the CPLEX environment, if necessary. */

if (env != NULL)
{
    status = CPXcloseCPLEX(&env);

    /*
       Note that CPXcloseCPLEX produces no output,
       so the only way to see the cause of the error is to use
       CPXgeterrorstring.  For other CPLEX routines, the errors will
       be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.
     */
    if (status)
    {
        if (opt_verbose>=1)
        {
            mexPrintf("Could not close CPLEX environment.\n");
            dispCPLEXerror(env, status);
        }
    }
    else
    {
    /* Just to be on the safe side we declare that the environment
    after closing is NULL. In this way we avoid possible error when
    trying to clear the same mex file afterwards. */
        env = NULL;
    }
}

if (!errors)
{
    *stat=0;
```

```
    }

    /* Pass computation to the real outputs and clear not used memory */

    STAT_OUT = STAT;

    /* Free allocated memory. */
    if (A_matbeg != NULL)
        mxFree(A_matbeg);
    if (A_matcnt != NULL)
        mxFree(A_matcnt);
    if (A_matind != NULL)
        mxFree(A_matind);
    if (A_matval != NULL)
        mxFree(A_matval);

    if (b_matval != NULL)
        mxFree(b_matval);

    if (H_matbeg != NULL)
        mxFree(H_matbeg);
    if (H_matcnt != NULL)
        mxFree(H_matcnt);
    if (H_matind != NULL)
        mxFree(H_matind);
    if (H_matval != NULL)
        mxFree(H_matval);

    if (f_matval != NULL)
        mxFree(f_matval);

    if (sense != NULL)
        mxFree(sense);

    if (nQC > 0)
    {
        int ii;
        for (ii=0; ii<nQC; ii++)
        {
            if (QC_linind[ii] != NULL)
                mxFree(QC_linind[ii]);
            if (QC_linval[ii] != NULL)
                mxFree(QC_linval[ii]);
            if (QC_quadrow[ii] != NULL)
                mxFree(QC_quadrow[ii]);
```

```
            if (QC_quadcol[ii] != NULL)
                mxFree(QC_quadcol[ii]);
            if (QC_quadval[ii] != NULL)
                mxFree(QC_quadval[ii]);
        }
    }
    if (QC_r != NULL)
        mxFree(QC_r);
    if (QC_linnzcnt != NULL)
        mxFree(QC_linnzcnt);
    if (QC_linind != NULL)
        mxFree(QC_linind);
    if (QC_linval != NULL)
        mxFree(QC_linval);
    if (QC_quadnzcnt != NULL)
        mxFree(QC_quadnzcnt);
    if (QC_quadrow != NULL)
        mxFree(QC_quadrow);
    if (QC_quadcol != NULL)
        mxFree(QC_quadcol);
    if (QC_quadval != NULL)
        mxFree(QC_quadval);
    if (LB_matval != NULL)
        mxFree(LB_matval);
    if (UB_matval != NULL)
        mxFree(UB_matval);
    if (intparcode != NULL)
        mxFree(intparcode);
    if (intparvalue != NULL)
        mxFree(intparvalue);
    if (doubleparcode != NULL)
        mxFree(doubleparcode);
    if (doubleparvalue != NULL)
        mxFree(doubleparvalue);
    if (vartype != NULL)
        mxFree(vartype);
    if (opt_x0i != NULL)
        mxFree(opt_x0i);
    if (opt_x0 != NULL)
        mxFree(opt_x0);
    if (!errors)
    {TROUBLE_mexErrMsgTxt("There were errors.");}
    return;
}
```

Example : Here is a small example to illustrate how to use this function to write a quadratic program in QP.lp file with .lp format.

```
H = [1 -1; -1 2];
f = [-2; -6];
A = [1 1; -1 2; 2 1];
b = [2; 2; 3];
lb = zeros(2,1);

STAT=CPXwriteprob('QP1.lp',H,f,A,b,[],[],lb);
```

A.3 General DCA prototype

In this section, we present a short MATLAB pseudocode providing a framework to the general DCA prototype. You can consult this format and modify the code to solve any DC programs with DCA. A short introduction to the Input and Output data structures, as well as some options are given in the head of the function "DCA".

```
function [output,fval,xopt] = DCA(DCP,x0,options)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DCA for solving DC problem defined as:
%
% minimize g(x)-h(x)
% s.t. x in C
%
% where g and h are both convex function, and C is a convex set.
%
% Output data:
% xopt: computed solution
% fval: computed optimal value
%
% output.time: Total computing time
% output.iter: DCA's iterations
% output.msg: informations about the solution state
% output.exitflag: the state of the result
%  (1: converge, 2: beyond the max iteration, 0: error)
%
% Input Data:
% DCP: A structure to define the DC problem.
%     You can define your own structure without any limite.
%     the DCP must consiste of all necessary data of the DC problem.
%     such as the number of variables, matrix, or polynomial functions etc.
%     You can also use YALMIP to define your variables and problems.
%
%    ex1: using Yalmip format
%    + define a variable with Yalmip
%    DCP.x = sdpvar(n,1);
%    + define fonctions with Yalmip
%    DCP.g = x'*H*x + u'*x;
%    DCP.h = x'*Q*x + v'*x;
%    + define constraints with Yalmip
%    DCP.C = [A*x<=b, x'*D*x +d'*x <=s, lb<=x<=ub];
%    + for some constants
%    DCP.n = 1000; % the number of variables
%
%    ex2: using matrix form
%    DCP.n = 1000; % the number of variables
```

```
%    DCP.g.H = H;
%    DCP.g.f = u;
%    DCP.h.H = Q;
%    DCP.h.f = v;
%    DCP.C.A = A;
%    DCP.C.b = b;
%    DCP.C.D = D;
%    DCP.C.d = d;
%    DCP.C.s = s;
%    DCP.C.lb = lb;
%    DCP.C.ub = ub;
%
% You can just define the elements of the DCP structure freely
% as you wish and as you need.
%
% x0: a given initial point.
%
% options:
% options.MaxDCA: max iterations of DCA.
% options.TolF: Tolerance to the objective function (to stop algorithme)
% options.TolXopt: Tolerance to the vector x (to stop algorithme)
% options.display: Display the verbose message during the computation
%
%  Authors:     Yi-Shuai NIU
%  Contact:     INSA de Rouen
%               niuys@insa-rouen.fr
%
%  History: date: 2008.01.01  ver2.1 revised
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%% Check input parameters %%%%%%%%%
if nargin>3
    error('Parameter error:','Two many input parameters');return;
end
if nargin<3
    % initialize options if not given
    options.MaxDCA=1000;
    options.TolF=1e-8;
    options.TolXopt=1e-6;
    options.display=1;
end

% checking the fields of options
if ~isfield(options,'MaxDCA')
    options.MaxDCA=1000;
```

```
end
if ~isfield(options,'TolF')
    options.TolF=1e-8;
end
if ~isfield(options,'TolXopt')
    options.TolXopt=1e-5;
end

% check initial point, give a random initial point if not given
if nargin >= 2
    if isempty(x0)
        x0=rand(DCP.n,1);
        i=1:DCP.n;
        lindex = x0(i)<lb(i);
        if any(lindex),
            x0(lindex)=lb(lindex);
        end
        i=1:n4;
        uindex = x0(i)>ub(i);
        if any(uindex)
            x0(uindex)=ub(uindex);
        end
    end
end


%%%%%%%%%%%%%% Display verbox information %%%%%%%%%%%%%%%%
if options.display==1
    msg=sprintf('------DC Problem descriptions:------\n
    Number of variables: %d ',DCP.n);
    disp(msg);
    msg=sprintf('------Parameters:------\n
    X tolerance: %e\n
    Function tolerance: %e\n'
    ,options.TolXopt,options.TolF);
    disp(msg);
    msg=sprintf('\t #Iter \t\t Obj \t\t error_X \t\t error_Obj\n');
    disp(msg);
    msg=sprintf('\t -------------------------------------------------\n');
    disp(msg);
end
%%%%%%%%%%%%%% Call DCA procedure %%%%%%%%%%%%%%%%
Iteration=0;
%
tic;
```

```
[output,fval,xopt]=DCA_PROC(DCP,x0,options,Iteration);
output.time=toc;
end
%%%%%%%%%%%%% END of DCA%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DCA procedure
%
% exitflag=0; Error
% exitflag=1; DCA converges
% exitflag=2; beyond the max iteration of DCA
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [output,f,X] = DCA_PROC(DCP,x0,options,Iteration)

xk = x0;

Iteration=Iteration+1;

% here, you should write your own function to compute yk from xk.
% ex: yk = grad(h)(xk)
yk = Computeyk(DCP,xk,options);

% here, you should write your own function to compute xk+1 from yk.
% ex: xk1 in argmin{g(x) - yk'*x: x in C}
[xk1,fk1,stat] = Computexk1(DCP,yk,options);

% checking if the problem is solved without error.
if stat.problem == -1
    msg=sprintf('Error during the iteration %d for computing xk+1\n
    solution state: %d',Iteration, stat.errorcode);
    disp(msg);
    X=[];
    f=[];
    output.exitflag=0; % error
    output.iter=Iteration;
    output.msg=msg;
    return;
end

% Evaluate the value of the DC objective function g-h at point xk and xk1.
fvalk=evalFunc(DCP,xk);
fvalk1=evalFunc(DCP,xk1);

deltax = norm(xk1-xk);
```

```
deltaf = abs(fvalk1-fvalk);
if options.display==1
    msg=sprintf('\t %d\t\t%.10f\t\t%.10f\t\t%.10f',Iteration,...
    fvalk1,deltax,deltaf);
    disp(msg);
end


% Check stopping conditions

if deltax<=options.TolXopt*(1+norm(xk)) ||
   deltaf<=options.TolF*(1+abs(fvalk))
    % or sometime use "if deltax<=options.TolXopt || deltaf<=options.TolF"
    msg='DCA Converges';
    X=xk1;
    f=fvalk1;
    output.exitflag=1;
    output.iter=Iteration;
    output.msg=msg;
    return;
elseif Iteration > options.MaxDCA
        msg=sprintf('\n Beyond the max iteration of DCA: %d\n
        Modify the option.MaxDCA for more iterations',options.MaxDCA);
        X=xk1;
        f=fvalk1;
        output.exitflag=2;
        output.iter=Iteration;
        output.msg=msg;
        return;
    end
    xk=xk1;
    [output,f,X]=DCA_PROC(DCP,xk,options,Iteration);
end
end
```

To use this routine, you need to write your MATLAB functions to compute y^k, x^{k+1}, and $(g-h)(x^k)$.

y^k can be often computed explicitly as

$$y^k \in \partial h(x^k)$$

The prototype could be given as :

```
function yk = Computeyk(DCP,xk,options)
% your code here
end
```

For computing x^{k+1} by solving the convex program :

$$x^{k+1} \in \arg\min\{g(x) - x^T y^k : x \in C\}$$

The MATLAB prototype is :

```
function [xk1,fk1,stat] = Computexk1(DCP,yk,options)
% your code here
end
```

Evaluating the value of the DC function $g - h$ at a given point is necessary, so you should write your MATLAB function as the prototype :

```
function fval=evalFunc(DCP,x)
% your code here
end
```

A.4 Prototype of DCA-BB

The DCA-BB algorithm is not unique. The structure of this kind of algorithm maybe quite different between each other, so we'd like to present here some important routines to show how to create and manage NODE structures during the branch and bound process. Note that this code is not an optimized one. It could be only considered as a simple example and simplified prototype of the DCA-BB.

```
function [x,fval,output] = PROCB_B(DCP,x0,options)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Branch and Bound process
%
% list: for storing existing nodes' index number.
%
% NODE structure:
% NODE.fval lower bound of the node
% NODE.co   convex hull function
% NODE.vertex  vertices set. a n x n matrix
% NODE.vetexval values of the convex hull function at the vertices.
% NODE.ub, NODE.lb box constraint of the node
% The node structure maybe different according to your specific problem.
%
% exitflag=0; error
% exitflag=1; optimized
%
%  Authors:     Yi-Shuai NIU
%  Contact:     INSA de Rouen
```

```
%               niuys@insa-rouen.fr
%
%  History: date: 2008.01.01  ver3.1 revised
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
IterBB=0;
nvar=DCP.DCFn.n; %number of variables
gapval=0.01; %the max gapval

%1. Initialize Upper bound (provided by DCA as an input parameters)
BESTUB.x = x0.x; % this must be a feasible point; [] otherwise
BESTUB.fval = x0.fval; % if x0.x is [], here should be +infty
UBSOL.x = BESTUB.x; UBSOL.fval = BESTUB.fval;

%2. Create root node
NODE.vetex=eye(nvar);
for i=1:nvar
    % compute vertices set and evaluate values
    NODE.vetexval(i) = FuncF(NODE.vetex(:,i),DCP.DCFn.Coef,DCP.DCFn.R,
    DCP.DCFn.V,DCP.DCFn.S, DCP.DCFn.K,DCP.DCFn.n)
    -0.5*DCP.DCFn.mu*NODE.vetex(:,i)'*NODE.vetex(:,i);
end
NODE.ub=DCP.ub;
NODE.lb=DCP.lb;
NODE.fval=-inf;
NODE.co.c=zeros(nvar,1); %co(-h)=cx+b
NODE.co.b=0;

his_nodenum=1; % total created node numeration
list=[1]; % list of current existing nodes

%3. Iterations
while ~isempty(list) % if list is not empty, continuous
    IterBB=IterBB+1;

    % get smallest lower bound node
    [minlb,ind_m]=min([NODE.fval]);
    if abs(BESTUB.fval - minlb) <= gapval
    % test the gap between the lower bound and best upper bound.
        display('stop!');
        break; % jump out the "while" cycle
    end
    list(ind_m)=[]; % pick out the node from the list index
    PROB = NODE(ind_m); % get the chosen node problem
    NODE(ind_m)=[]; % pick out the node from the node list
```

```
% Delate all nodes whose lower bounds are bigger or equal
% to the current best upper bound.
dellist=[];
for i=1:length(list)
    if NODE(i).fval > BESTUB.fval
        dellist=[dellist,i];
    end
end
list(dellist)=[];
NODE(dellist)=[];

% Branching - computing vertices
[SUBPROB(1),SUBPROB(2)] = SeparateNode(PROB,DCP);

% Construct convex hull functions
SUBPROB(1).co = envelopconvex(SUBPROB(1).vetex,SUBPROB(1).vetexval);
SUBPROB(2).co = envelopconvex(SUBPROB(2).vetex,SUBPROB(2).vetexval);

% Solving the two subproblems
% SUBPROB1
if ~isempty(SUBPROB(1).co)
% calling cplex11 to solve the subproblem
[SOLUTIONS(1).x,SOLUTIONS(1).fval,SOLUTIONS(1).exitflag] = ...
    callcplex11(Q,SUBPROB(1).co.c,DCP.A,DCP.b,DCP.Aeq,DCP.beq,
    SUBPROB(1).lb,SUBPROB(1).ub);
else
    SOLUTIONS(1).exitflag=-1; % problem is infeasible
end

% SUBPROB1
if ~isempty(SUBPROB(1).co)
[SOLUTIONS(2).x,SOLUTIONS(2).fval,SOLUTIONS(2).exitflag] = ...
    callcplex11(Q,SUBPROB(2).co.c,DCP.A,DCP.b,DCP.Aeq,DCP.beq,
    SUBPROB(2).lb,SUBPROB(2).ub);
else
    SOLUTIONS(2).exitflag=-1; % problem is infeasible
end

% Check the solutions.
%1 check feasibility (delete)
%2 check whether the lower bound >= the best upper bound (delete)
%3 check if we get a better feasible point.
%4 if we obtain a better feasible point, then update upper bound.
% (option: we can restart DCA here).
%5 Add the problem into the list of nodes.
```

```
    for i=1:2
        if SOLUTIONS(i).exitflag == 1 %1. check feasibility
            SOLUTIONS(i).fval=SOLUTIONS(i).fval+SUBPROB(i).co.b;
            %2. check the lower bound
            if SOLUTIONS(i).fval < BESTUB.fval
                curval=FuncF(SOLUTIONS(i).x,DCP.DCFn.Coef,DCP.DCFn.R,
                DCP.DCFn.V,DCP.DCFn.S,DCP.DCFn.K,DCP.DCFn.n);
                %3. check if we get a better feasible point
                if curval < BESTUB.fval
                %4. Restart DCA
                    [output,fval,x]=RestartDCA (DCP,SOLUTIONS{i}.x);
                    % Here, we can restart DCA
                    % check if DCA find a better feasible upper bound
                    % and update the best upper bound if necessary
                    if checkfeasibility(DCP,x)
                        BESTUB.fval = fval;
                        BESTUB.x =x;
                    else
                        BESTUB.fval=SOLUTIONS{i}.fval;
                        BESTUB.x = SOLUTIONS{i}.x;
                    end
                    UBSOL(end+1).x=BESTUB.x;
                    UBSOL(end).fval=BESTUB.fval;
                end
                %5. Add the problem into the list of nodes
                his_nodenum = his_nodenum+1;
                SUBPROB(i).fval = SOLUTIONS(i).fval;
                NODE(end+1)=SUBPROB(i);
                list(end+1)=his_nodenum;
            end
        end
    end
end

disp('Terminate');
if ~isempty(BESTUB.x)
    % optimized
    x=BESTUB.x;
    fval= BESTUB.fval;
    exitflag=1;
    output.IterBB=IterBB;
    return;
else
    % no solution found (maybe the problem is infeasible)
    x=[];
```

```
    fval=[];
    exitflag=0;
    output.IterBB=IterBB;
    return;
end
end
```

Chapitre 9

Conclusion et Perspectives

Cette thèse s'intéresse particulièrement aux études théorique et algorithmique de la programmation DC et DCA, en les combinant aux techniques de relaxation DC/SDP, et Séparation et Evaluation (SE) pour résoudre plusieurs types de problèmes importants en optimisation non convexe (notamment en Optimisation Combinatoire et Optimisation Polynomiale).

Pour chacun de ces problèmes, nous avons proposé de nouveaux algorithmes (basés sur DCA) adaptés à leur structure particulière et exploité l'efficacité des décompositions DC et des stratégies d'initialisation. La structure spéciale de chaque problème a été étudiée avec soin pour trouver des approches bien adaptées et peu coûteuses. DCA joue un rôle crucial pour les algorithmes proposés. Les techniques de formulation et reformulation permettent de transformer les problèmes non convexes étudiés en programmes DC. On peut ensuite utiliser DCA pour les résoudre. Les résultats des simulations numériques montrent les performances remarquables de nos approches.

Nos contributions portent essentiellement sur les trois types de problèmes suivants :

1. La programmation mixte avec variables entières.
   Nous avons proposé une technique de reformulation de variable entière en utilisant des fonctions continues et/ou différentiables (formulation continue) sans avoir besoin de remplacer les variables entières par des variables binaires (formulation binaire). Cette nouvelle technique a fait preuve de beaucoup plus d'efficacité que la formulation binaire car celle ci ne requiert aucune variable additionnelle. Cette formulation permet également de trouver immédiatement une reformulation DC qui est ensuite résolue efficacement avec l'algorithme DCA. En combinant DCA avec SE, nous avons proposé de nouveaux algorithmes pour résoudre globalement le programme mixte avec variables entières. Les résultats numériques ont été comparés à ceux de CPLEX pour les programmes linéaires et quadratiques convexes mixtes.
2. La programmation polynomiale.
   Une technique de décomposition DC pour la fonction polynomiale homogène d'ordre k a été étudiée. Ceci permet de formuler un programme mathématique dont la fonction objectif est polynomiale homogène sous contraintes convexes en un programme DC. Cette technique a été appliquée avec succès à la résolution du problème de gestion de portefeuille avec moments d'ordre supérieur (Portfolio selection with Higher Moments). Une nouvelle approche d'optimisation globale, combinant DCA aux techniques de relaxation DC et un schéma de SE, est proposée. Sur le plan numérique, nous avons comparé les résultats fournis par DCA à

ceux fournis par plusieurs approches locales et globales existantes (une méthode du point intérieur, une méthode de région de confiance, le logiciel LINGO, et Gloptipoly etc.). On a également présenté nos travaux préliminaires portant sur la résolution du programme polynomial général via les approches de la programmation DC en combinant DCA avec les relaxations DC/SDP et les techniques de SE etc.

3. La programmation sous contraintes de matrices semi-définies
   Concernant le programme sous contraintes semi-définies, nous nous sommes consacrés au problème de réalisabilité des contraintes BMI (Bilinear Matrix Inequality) et QMI (Quadratic Matrix Inequality) en contrôle optimal. Ce genre de contraintes peut être considéré comme une généralisation des contraintes bilinéaires et quadratiques. Nous avons étudié les techniques de reformulation de BMI et QMI en programme DC afin de proposer de nouvelles approches basées sur DCA pour résoudre ce type de problème. Les résultats issus des simulations numériques effectuées avec nos algorithmes, et comparés à celles effectuées avec le logiciel PENBMI sont également présentés.

Enfin, une bonne partie concernant le codage et l'implémentation logicielle des différents algorithmes en plusieurs langages de programmation (C/C++, MATLAB, E-language ...) permet de confirmer l'utilisation pratique et d'enrichir nos travaux de recherche.

Perspectives

1. Programmation quadratique non convexe mixte en variables entières. Nous avons proposé des approches portant sur la résolution de programmes linéaires et quadratiques convexes en variables mixtes entières dans la première partie. Ces algorithmes ne peuvent être appliqués directement au programme quadratique non convexe mixte avec variables entières. Grâce aux travaux préliminaires pour la résolution du programme quadratique présentés dans la deuxième partie, une approche DCA peut être proposée immédiatement à l'aide de la reformulation SDP.
2. On peut proposer de développer des approches de la programmation DC pour la résolution du problème d'optimisation sous contraintes BMI, QMI, et ainsi PMI (Polynomial Matrix Inequality).
3. En ce qui concerne la programmation polynomiale avec contraintes de fonctions polynomiales homogènes, si les fonctions polynomiales homogènes interviennent dans la formulation des contraintes, alors les contraintes deviennent non convexes. La technique de décomposition DC dans le Chapitre 6 permet de ramener ce programme non convexe à un problème de minimisation d'une fonction DC sous contraintes DC qui est résoluble dans le cadre de la programmation DC.

4. Il est intéressant de poursuivre les recherches préliminaires abordées dans le Chapitre 7 (les approches de la programmation DC pour résoudre le programme polynomial général). Par exemple, pour résoudre des problèmes à grande dimension, DCA présenté dans ce chapitre exige la résolution d'un programme SDP à chaque itération. Or les méthodes existantes (pour résoudre un programme SDP) ne peuvent traiter efficacement un problème de très grande dimension. Cette caractéristique limite la taille du problème résoluble. La manière de surmonter cette difficulté doit encore être étudiée.
5. Récemment, nous avons aussi étudié plusieurs autres problèmes comme "La programmation mathématique sous contraintes d'inégalité variationnelle affine". Ce problème est équivalent à un programme sous contraintes de complémentarité linéaires. Nous avons réussi à appliquer nos approches DCA combinées aux techniques de reformulation SDP pour résoudre ce type de problème. Par ailleurs, nous avons aussi appliqué nos approches DC au programme mixte pour résoudre un problème d'optimisation robuste de gestion de portefeuille sous contraintes de cardinalit. Les articles sont en cours de rédaction.
6. On peut envisager de poursuivre le développement et l'amélioration du code de DCA pour créer un groupe de "solveurs DCA" professionnel et de plus haute performance.

Toutes ces questions feront l'objet de nos travaux dans un futur proche.

Bibliographie

[1] Markowitz H.M. : Portfolio Selection, Journal of Finance, Vol 7 No 1, 77-91, (1952)

[2] Steinbach M.C. : Markowitz Revisited : Mean-Variance Models in Financial Portfolio Analysis. Society for Industrial and Applied Mathematics, Vol 43 No 1, 31-85, (2001)

[3] Zenios S.A., Jobst N.J. : The Tail that Wags the Dog : Integrating Credit Risk in Asset Portfolios. Journal of Risk Finance, Fall 2001, 31-44, (2001)

[4] Harvey C.R., Siddique A. : Conditional Skewness in Asset Pricing Tests. The Journal of Finance, Vol 55(3), 1263-1295, (2000)

[5] Arditti F.D., Levy H. : Portfolio Efficiency Analysis in Three Moments : The Multiperiod Case. Journal of Finance, American Finance Association, Vol. 30(3), 797-809, (1975)

[6] Rockinger M., Jondeau E. : Conditional Volatility, Skewness and Kurtosis : Existence and Persistence and Comovements. Journal of Economic Dynamics and Control, Vol 27, 1699-1737, (2003)

[7] Jean W.H. : The extension of portfolio analysis to three or more parameters. Journal of Financial and Quantitative. Analysis 6, 505-515, (1971)

[8] Athayde G., Flores R. : Finding a maximum skewness portfolio. Society for Computational Economics, 271, (2001)

[9] Harvey C.R., Liechty J.C., Liechty M.W. and Mueller P. : Portfolio Selection With Higher Moments. Duke University, Working Paper, (2003)

[10] Hoang T., Concave programming under linear constraints, Translated Soviet Mathematics, Vol. 5, 1437-1440, (1964)

[11] Konno H., Phan Thien T., Hoang T., Optimization on low rank nonconvex structures, Kluwer Academic Publishers, (1997)

[12] Aleksandrov A.D. : On surfaces represented as the difference of convex functions. Izvestiya Akad. Nauk Kazah. SSR. 60, Ser. Mat. Meh. 3, (1949)

[13] Hartman P. : On functions representable as a difference of convex functions. Pacific J. Math. 9, 707-713, (1959)

[14] Pham Dinh T. : Algorithmes de calcul du maximum des formes quadratiques sur la boule unité de la norme du maximum, Séminaire d'analyse numérique, Grenoble, No 247, (1976)

[15] Pham Dinh T. : Contribution à la théorie de normes et ses applications à l'analyse numérique. Thèse de Doctorat d'Etat Es Science, Université Joseph Fourier-Grenoble, (1981)

[16] Pham Dinh T. : Convergence of subgradient method for computing the bound norm of matrice. Linear Alg. and its Appl. 62, 163-182, (1984)

[17] Pham Dinh T. : Algorithmes de calcul d'une forme quadratique sur la boule unité de la norme maximum, Numer. Math. 45, 377-440, (1985)

[18] Pham Dinh T. : Algorithms for solving a class of non convex optimization problems. Methods of subgradients. Fermat days 85. Mathematics for Optimization, J.B. Hiriart Urruty (ed.), Elsevier Science Publishers, B.V. North-Holland, (1986)

[19] Pham Dinh T. : Duality in d.c. (difference of convex functions) optimization. Subgradient methods. Trends in Mathematical Optimization, K.H. Hoffmann et al. (ed.), International Series of Numer Math. 84, Birkhauser, (1988)

[20] Pham Dinh Tao and Le Thi Hoai An, Stabilité de la dualité lagrangienne en optimisation d.c. (différence de deux fonctions convexes). C.R.Acad. Paris, t.318,SérieI, (1994)

[21] Le Thi H.A. : Analyse numérique des algorithmes de l'optimisation d.c. approches locales et globales. Code et simulations numériques en grande dimension. Applications, Thèse de Doctorat de l'Université de Rouen, (1994)

[22] Thai Quynh P., Le Thi H.A., Pham Dinh T. : On the global solution of linearly constrained indefinite quadratic minimization problems by decomposition branch and bound method. RAIRO, Recherche Opérationnelle, Vol. 30 (1), 31-49, (1996)

[23] Pham Dinh T., Le Thi H.A., Convex analysis approach to D.C. programming : Theory, Algorithms and Applications. Acta Mathematica Vietnamica, Vol.22 No.1, 289-355, (1997)

[24] Le Thi H.A., Pham Dinh T. : Solving a class of linearly constrained indefinite quadratic problems by DC Algorithms. Journal of Global Optimization, Vol. 11, 253-285, (1997)

[25] Pham Dinh T., Le Thi H.A. : DC optimization algorithms for solving the trust region subproblem, SIAM J.Optimization, Vol. 8, 476-507, (1998)

[26] Le Thi H.A., Pham Dinh T., Le Dung M. : Exact penalty in d.c. programming. Vietnam Journal of Mathematics, Vol. 27(2), 169-178, (1999)

[27] Le Thi H.A. : An efficient algorithm for globally minimizing a quadratic function under convex quadratic constraints. Mathematical Programming, Ser. A, Vol. 87(3), 401-426, (2000)

[28] Le Thi H.A., Pham Dinh T. : A continuous approach for large-scale constrained quadratic zero-one programming. (In honor of Professor ELSTER, Founder of the Journal Optimization), Optimization Vol. 45(3), 1-28, (2001)

[29] Pham Dinh T., Le Thi H.A. : DC Programming. Theory, Algorithms, Applications : The State of the Art. First International Whorkshop on Global Constrained Optimization and Constraint Satisfaction, Nice, October 2-4, (2002)

[30] Le Thi H.A. : Solving large scale molecular distance geometry problems by a smoothing technique via the gaussian transform and d.c. programming, Journal of Global Optimization, Vol. 27(4), 375-397, (2003)

[31] Le Thi H.A., Pham Dinh T. : Large Scale Molecular Optimization From Distance Matrices by a D.C. Optimization Approach. SIAM Journal on Optimization, Vol. 4(1), 77-116, (2003).

[32] Pham Dinh T., Le Thi H.A. : The DC programming and DCA Revisited with DC Models of Real World Nonconvex Optimization Problems. Annals of Operations Research, Vol.133, 23-46, (2005)

[33] H. A. Le Thi, N.V. Vinh, et T. Pham Dinh : A Combined DCA and New Cutting Plane Techniques for Globally Solving Mixed Linear Programming, SIAM Conference on Optimization, Stockholm (2005).

[34] Le Thi H.A., Pham Dinh T., François A. : Combining DCA and Interior Point Techniques for large-scale Nonconvex Quadratic Programming. Optimization Methods & Software, Vol. 23(4), 609-629, (2008)

[35] Le Thi H.A., Moeini M., Pham Dinh T. : DC Programming Approach for Portfolio Optimization under Step Increasing Transaction Costs. Optimization, Volume 58, Issue 3, 267-289, (2009)

[36] Le Thi H.A., Moeini M., Pham Dinh T. : DC Programming Approach for Portfolio Optimization under Step Increasing Transaction Costs. Optimization, Volume 58, Issue 3, 267-289, (2009)

[37] Le Thi H.A., Moeini M., Pham Dinh T. : Portfolio Selection under Downside Risk Measures and Cardinality Constraints based on DC Programming and DCA. Computational Management Science, Issue 4, (2009)

[38] Nguyen Canh Nam, Le Thi Hoai An, Pham Dinh Tao : A Branch and Bound Algorithm Based on DC Programming and DCA for Strategic Capacity Planning in Supply Chain Design for a New Market Opportunity. Operations research proceedings (2006)

[39] Le Thi Hoai An, Pham Dinh Tao, Nguyen Van Thoai, Nguyen Canh Nam, D.C. Optimization Techniques for Solving a Class of Nonlinear Bilevel Programs. Journal of Global Optimization, Vol 44, Num 3, pp 313-337, (2009)

[40] Pham Dinh T., Nguyen Canh N. and Le Thi H.A., DC Programming and DCA for Globally Solving the Value-At-Risk, to appear in Computational Management Science Issue 4, (2009)

[41] Pham Dinh T., Nguyen Canh N. and Le Thi H.A., An efficient combined DCA and B&B using DC/SDP relaxation for globally solving binary quadratic programs. Journal of Global Optimization, Online first, 12 December, (2009)

[42] Niu Y.S., Pham Dinh T. : A DC Programming Approach for Mixed-Integer Linear Programs. Computation and Optimization in Information Systems and Management Sciences, Book Series : Communications in Computer and Information Science, Springer Berlin Heidelberg, 244-253, (2008)

[43] Le Thi H.A., Pham Dinh T., Huynh Van N. : Exact penalty techniques in DC programming. Technical report, National Institute for Applied Sciences - Rouen, France, (2007)

[44] Le Thi H.A., Pham Dinh T., Huynh Van N. : Exact penalization and error bounds for inequality systems of concave functions and of nonconvex quadratic functions. Technical report, National Institute for Applied Sciences - Rouen, France, (2008)

[45] Le Thi H.A., Huynh Van N., Pham Dinh T. : Convergence Analysis of DC Algorithm for DC programming with subanalytic data. Technical report, National Institute for Applied Sciences - Rouen, France, (2009)

[46] Hiriart-Urruty J.B. : Generalized differentiability, duality and optimization for problems dealing with differences of convex functions, in Convexity and Duality in Optimization. Lecture Notes in Economics and Mathematical Systems, Vol. 256, 37-70, (1986)

[47] Rockafellar R.T. : Convex Analysis. Princeton University Press, N.J., (1970)

[48] Hiriart-Urruty J.B., Lemarechal C. : Convex Analysis and Minimization Algorithms, Springer, Berlin, (1993)

[49] Horst R., Hoang T. : Global Optimization : Deterministic Approaches, 2nd revised edition. Springer, Berlin, (1993)

[50] Hoang T. : Convex Analysis And Global Optimization. Kluwer Academic Publishers, (1998)

[51] Horst R., Pardalos P.M., Thoai N.V. : Introduction to Global Optimization - Second Edition. Kluwer Academic Publishers, Netherlands, (2000)

[52] Toland J.F. : Duality in Nonconvex Optimization. J. Mathematical Analysis and Applications, 58, 415-428, (1978)

[53] Wolkowicz H., Saigal R., Vandenberghe L. : Handbook of Semidefinite Programming - Theory, Algorithms, and Applications. Kluwer Academic Publishers, USA, (2000)

[54] Atteia M., Elqortobi A. : Quasi-convex duality, in A. Auslender et al. (eds.), Optimization and Optimal Control, Proc. Conference Oberwolfach March 1980, Lecture notes in Control and Inform. Sci. 30, 3-8, Springer-Verlag, Berlin, (1981)

[55] Penot J.P, Duality for anticonvex programs, Journal of Global Optimization archive. Volume 19, Issue 2, 163-182, (2001)

[56] Parpas P., Rustem B. : Global optimization of the scenario generation and portfolio selection problems, Computational Science and Its Applications, Lecture Notes in Computer Science, Vol. 3982, 908-917, (2006)

[57] Lai K.K, Yu L., Wang S.Y. : Mean-Variance-Skewness-Kurtosis-based Portfolio Optimization. Proceedings of the First International Multi-Symposiums on Computer and Computational Sciences Volume 2 (IMSCCS'06), 292-297, (2005)

[58] Gondran M., Minoux M. : Graphes et algorithmes, 250-251. Eyrolles, Paris (1995)

[59] Frank J.F., Sergio M.F., Petter N.K. : Financial Modeling of the Equity Market : From CAPM to Cointegration, 131-139. Wiley, USA, (2006)

[60] Lasserre J.B. : Global optimization with polynoials and the problem of moments. SIAM J. Optim, Vol. 11(3), 796-817, (2001)

[61] Henrion D., Lasserre J.B., L̈fberg J. : GloptiPoly 3 : moments, optimization and semidefinite programming, Version 3.4, 30 september, (2008)

[62] Henrion D., Lasserre J.B. : GloptiPoly : global optimization over polynoials with Matlab and SeDuMi. ACM Transactions on Mathematical Software, Vol 29(2), 165-194, (2003)

[63] Vandenberghe L., Boyd S. : Semidefinite Programming. SIAM Review 38, 49-95, (1996)

[64] Wolkowicz H., Saigal R., Vandenberghe L.(editors) : Handbook on Semidefinite Programming. Kluwer, (2000)

[65] Helmberg C. : Semidefinite Programming for Combinatorial Optimization. Habilitationsschrift, TU Berlin, (2000)

[66] Todd M.J. : Semidefinite optimization. Acta Numerica 10, 515-560, (2001)

[67] Robert M. Freund : Introduction to Semidefinite Programming (SDP). Massachusetts Institute of Technology, (2004)

[68] C. Helmberg, F. Rendl, and R. Weismantel : A Semidefinite Programming Approach to the Quadratic Knapsack Problem, J. of Combinatorial Optimization, Vol. 4, No. 2, pp. 197-215, (2000)

[69] Balas E. : Disjunctive programming : cutting planes from logical conditions. In : Mangasarian, O.L., Meyer, R.R., Robinson, S.M.(eds.), Nonlinear Programming 2, Academics Press, New York, 279-312 (1975)

[70] Sherali H.D., Adams W.P. : A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems. SIAM J. Discrete Math, 3 (3), 411-430, (1990)

[71] M. Bauvin, M. Goemans : The quadratic knapsack problem-a survey. Discrete Applied Mathematics, Volume 155, Issue 5, 623-648, (2007)

[72] Lovasz L. and Schrijver A. : Cones of matrices and set-functions and 0-1 optimization. SIAM J. Optim., 1, 166-190, (1991)

[73] Borchers B. and J. G. Young J.G. : Implementation of a primal¨Cdual method for SDP on a shared memory parallel architecture. Computational Optimization and Applications 37(3) :355-369, 2007

[74] Benson S.J., Ye Y.Y., Zhang X. : Solving Large-Scale Sparse Semidefinite Programs for Combinatorial Optimization, SIAM Journal on Optimization, 10(2), pp. 443-461, (2000)

[75] Horst R. : A general class of branch-and-bound methods in global optimization with some new approaches for concave minimization. J. Optim. Theory Appl. 51, no. 2, 271-291, (1986)

[76] Horst R. : Deterministic global optimization with partition sets whose feasibility is not known : application to concave minimization, reverse convex constraints, DC-programming, and Lipschitzian optimization. J. Optim. Theory Appl. 58 ,no. 1, 11-37, (1988)

[77] Muu L.D., Phong T.Q., Pham Dinh T. : Decomposition methods for solving a class of nonconvex programming problems dealing with bilinear and quadratic function. Compuational Optimization and Applications 4, 203-216, (1995)

[78] Thoai N.V., Tuy H. : Convergent Algorithms for Minimizing a Concave Function. MATHEMATICS OF OPERATIONS RESEARCH Vol. 5, No. 4, 556-566, (1980)

[79] Luenberger D.G. : Linear and Nonlinear Programming. Second edition, pp. 366-380. Springer (2003)

[80] Ge R.P., Huang C.B. : A Continuous Approach to Nonlinear Integer Programming. Applied Mathematics and Computation, vol. 34, 39-60 (1989)

[81] Zhang G.Q., A note on "A Continuous Approach to Nonlinear Integer Programming". Applied Mathematics and Computation, vol 215, Issue 6, pp. 2388-2389, 15 November (2009)

[82] Nesterov Y., Nemirovskii A. : Interior-point polynomial methods in convex programming. Studies in Applied Mathematics, Vol. 13, Society for Industrial and Applied Mathematics, Philadelphia, PA, (1994).

[83] K. Fujisawa, M. Kojima and K. Nakata : SDPA (SemiDefinite Programming Algorithm) - user's manual - version 6.20. Research Report B-359, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, Tokyo, Japan, January 2005, revised May (2005). Available at `http://sdpa.indsys.chuo-u.ac.jp/sdpa/download.html`

[84] Tuan H.D., Hosoe S., Tuy H. : D.C. optimization approach to robust controls : Feasibility problems, IEEE Transactions on Automatic Control, Vol. 45, 1903-1909, (2000)

[85] Goh. K.C., Safonov M.G., Papavassilopoulos G.P. : Global optimization for the biaffine matrix inequality problem. Journal of Global Optimization, Vol. 7, 365-380, (1995)

[86] Goh K.C., Safonov M.G., and Ly J. H. : Robust synthesis via bilinear matrix inequalities. International Journal of Robust and Nonlinear Control, Vol. 6, 1079-1095, (1996)

[87] Sherali H.D., Alameddine A.R. : A new reformulation-linearization technique for bilinear programming problems. Journal of Global Optimization, Vol. 2, 379-410, (1992)

[88] Toker O., Özbay H. : On the NP-hardness of solving bilinear matrix inequalities and simultaneous stabilization with static output feedback. American Control Conference, Seattle, WA, (1995)

[89] Safonov M.G., Goh K.C., Ly J.H. : Control system synthesis via bilinear matrix inequalities. In Proceedings of the American Control Conference, Baltimore, MD, June (1994)

[90] VanAntwerp J.G. : Globally optimal robust control for systems with time-varying nonlinear perturbations. Master thesis, University of Illinois at Urbana-Champaign, Urbana, IL, (1997)

[91] Fujioka H., Hoshijima K. : Bounds for the BMI eingenvalue problem - a good lower bound and a cheap upper bound. Transactions of the Society of Instrument and Control Engineers, Vol. 33, 616-621, (1997)

[92] Fukuda M., Kojima M. : Branch-and-Cut Algorithms for the Bilinear Matrix Inequality Eigenvalue Problem. Computational Optimization and Applications, Vol. 19, Issue 1, 79-105, (2001)

[93] Kawanishi M., Sugie T., Kanki H. : BMI global optimization based on branch and bound method taking account of the property of local minima. In Proceedings of the Conference on Decision and Control, San Diego, CA, December (1997)

[94] Takano S., Watanabe T., Yasuda K. : Branch and bound technique for global solution of BMI. Transactions of the Society of Instrument and Control Engineers, Vol. 33, 701-708, (1997)

[95] Liu S.M., Papavassilopoulos G.P. : Numerical experience with parallel algorithms for solving the BMI problem. 13th Triennial World Congress of IFAC, San Francisco, CA, July (1996)

[96] Mesbahi M., Papavassilopoulos G.P. : A cone programming approach to the bilinear matrix inequality problem and its geometry. Mathematical Programming, Vol. 77, 247-272, (1997)

[97] Floudas C. A., Visweswaran V. : A primal-relaxed dual global optimization approach. Journal of Optimization Theory and Applications, Vol. 78, 187-225, (1993)

[98] Beran E.B., Vandenberghe L., Boyd S. : A global BMI algorithm based on the generalized Benders decomposition. In Proceedings of the European Control Conference, Brussels, Belgium, July (1997)

[99] Tuan H.D., Apkarian P., Nakashima Y. : A new Lagrangian dual global optimization algorithm for solving bilinear matrix inequalities. International Journal of Robust and Nonlinear Control, Vol. 10, 561-578, (2000)

[100] GloptiPoly 3 : http://www.laas.fr/~henrion/software/gloptipoly3/

[101] SeDuMi 1.2 : http://sedumi.ie.lehigh.edu/

[102] YALMIP : A Toolbox for Modeling and Optimization in MATLAB, Löfberg J. In Proceedings of the CACSD Conference, Taipei, Taiwan, (2004). http://control.ee.ethz.ch/~joloef/wiki/pmwiki.php

[103] LINGO 8.0 : LINDO Systems - Optimization Software : Integer Programming, Linear Programming, Nonlinear Programming, Global Optimization. http://www.lindo.com/

[104] CPLEX 11.0 : ILOG CPLEX Documentation, ILOG IBM. (2007) http://www-01.ibm.com/software/integration/optimization/cplex/

[105] CPLEX 12.1 : ILOG CPLEX Documentation, ILOG IBM. (2009) http://www-01.ibm.com/software/integration/optimization/cplex/

[106] Mosek 6.0 : The MOSEK optimization tools manual. (2009) http://www.mosek.com

[107] Tomlab : The TOMLAB Optimization Environment for fast and robust large-scale optimization in MATLAB. Avaible at http://tomopt.com/tomlab/

[108] COIN-OR : Computational Infrastructure for Operations Research - open source for the operations research community. http://www.coin-or.org/

[109] MIPLIB 3.0 Problem Set. http://www.caam.rice.edu/~bixby/miplib/miplib_prev.html

[110] MIPLIB2003 : Operations Research Letters, Vol. 34 , no 4, 361-372, (2006). http://miplib.zib.de/

[111] MATLAB R2008a : Documentation and User Guides. http://www.mathworks.com/

Résumé : L'objectif de cette thèse porte sur des recherches théoriques et algorithmiques d'optimisation locale et globale via les techniques de programmation DC & DCA, Séparation et Evaluation (SE) ainsi que les techniques de relaxation DC/SDP, pour résoudre plusieurs types de problèmes d'optimisation non convexe (notamment en Optimisation Combinatoire et Optimisation Polynomiale). La thèse comporte quatre parties :

La première partie présente les outils fondamentaux et les techniques essentielles en programmation DC & l'Algorithme DC (DCA), ainsi que les techniques de relaxation SDP, et les méthodes de séparation et évaluation (SE).

Dans la deuxième partie, nous nous intéressons à la résolution de problèmes de programmation quadratique et linéaire mixte en variables entières. Nous proposons de nouvelles approches locales et globales basées sur DCA, SE et SDP. L'implémentation de logiciel et des simulations numériques sont aussi étudiées.

La troisième partie explore des approches de la programmation DC & DCA en les combinant aux techniques SE et SDP pour la résolution locale et globale de programmes polynomiaux. Le programme polynomial avec des fonctions polynomiales homogènes et son application à la gestion de portefeuille avec moments d'ordre supérieur en optimisation financière ont été discutés de manière approfondie dans cette partie.

Enfin, nous étudions dans la dernière partie un programme d'optimisation sous contraintes de type matrices semi-définies via nos approches de la programmation DC. Nous nous consacrons à la résolution du problème de réalisabilité des contraintes BMI et QMI en contrôle optimal.

L'ensemble de ces travaux a été implémenté avec MATLAB, C/C++ ... nous permettant de confirmer l'utilisation pratique et d'enrichir nos travaux de recherche.

Mots clés : Optimisation locale et globale, Programmation DC, DCA, Séparation et Evaluation (SE), Relaxation Semi-définie, Relaxation DC, SDP, Programmation mixte en variables entières, Optimisation polynomiale, Polynômes homogènes, BMI, QMI, Implémentation logicielle.

Abstract : The main objective of this thesis focuses on theoretical and algorithmic researches of local and global optimization techniques to DC programming & DCA with Branch and Bound (B&B) and the DC/SDP relaxation techniques to solve several types of non-convex optimization problems (including Combinatorial Optimization and Polynomial Optimization). This thesis is divided into four parts :

We present in the first part some fondamental theorems and essential techniques in DC programming & DC Algorithm (DCA), the SDP Relaxation techniques, as well as the Branch and Bound methods (B&B).

In the second part, we are interested in solving mixed integer quadratic and linear programs. We propose new local and global approaches based on DCA, B&B and SDP. The implementation of software and numerical simulations have also been investigated.

The third part explores the DC programming approaches & DCA combined with a B&B technique and SDP for locally and globally solving a class of polynomial programming. The polynomial program with homogeneous polynomial functions and its application to portfolio selection problem involving higher order moments in financial optimization have been deeply studied in this part.

Finally, in the last part, we present our research on optimization problems under constraints of semi-definite matrices via our DC programming approaches. This part is dedicated to the resolution of the BMI and QMI feasibility problems in the field of optimal control.

All these proposed methods have been implemented with MATLAB, C++ etc., that allowing us to confirm the practical use and enrich our research works.

Keywords : DC programming, DCA, Branch and Bound (B&B), Semi-definite relaxation, SDP, Local and Global Optimization, Mixed-integer programming, Polynomial optimization, Homogeneous polynomial function, BMI, QMI, Software implementation.