

MASTER 2 Recherche  
INTELLIGENCE ARTIFICIELLE ET DÉCISION  
Université Pierre et Marie Curie (Paris VI)

Rapport de Stage

# Optimisation en nombres entiers de fonctions quadratiques non convexes soumises à des contraintes linéaires

Amélie Lambert

Directeurs de Stage

M. Alain Billionnet, professeur des universités  
Mme Sourour Elloumi, maître de Conférences

Laboratoire CEDRIC du Conservatoire National des Arts et Métiers

Soutenu le 11 Septembre 2006 devant le jury

M Patrice Perny, professeur des universités  
M Michel Minoux, professeur des universités  
M. Alain Billionnet, professeur des universités  
M Philippe Chrétienne, professeur des universités  
Mme Safia Kedad-Sidhoum, maître de Conférences

Laboratoire LIP6 de l'Université Paris VI

## Résumé

Cette étude traite de programmes mathématiques quadratiques non convexes à variables entières et dont la fonction objectif est soumise à des contraintes linéaires. Cette catégorie de problèmes, qui fait partie des problèmes difficiles, ne peut être résolue par un solveur standard sans transformation préalable. En effet, ces solveurs ne peuvent résoudre que des programmes où la fonction objectif est soit linéaire, soit quadratique et convexe. Nous aborderons donc différentes méthodes de résolution de cette catégorie de problèmes, tout d'abord grâce à des reformulations convexes, puis en comparant la qualité des différentes résolutions avec celle d'une linéarisation. Enfin, nous essayerons d'introduire une nouvelle idée de convexification, inspirée d'un mélange de ces dernières, qui serait plus performante. Pour évaluer la qualité de ces reformulations, nous avons réalisé un logiciel nommé **Resolution** qui est capable à la fois de générer et de lire des instances, mais aussi de les résoudre par la méthode souhaitée grâce à ses différentes options.

## Remerciements

Je tiens tout d'abord à remercier Sourour Elloumi et Alain Billionnet, qui m'ont encadré et guidé tout au long de mon stage. Leur attention et leurs conseils m'ont permis d'avancer dans mon travail et m'ont donné envie de poursuivre mon cursus en thèse avec eux.

Je remercie également Marie-Christine Costa et Christophe Picouveau pour leur accueil au sein du CEDRIC, ainsi qu'Agnès Plateau pour avoir partagé son bureau avec moi.

Enfin, je voudrais remercier les doctorants du CEDRIC, Nicolas Derhy, Marie-Christine Plateau, Cédric Bentz et Aurélie Le Maître, que j'ai côtoyé au cours de mon stage, pour leur disponibilité.

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Résolution d'un programme quadratique en nombres entiers par transformation en un programme quadratique en variables 0-1</b>	<b>7</b>
2.1	Reformulation du problème en variables 0-1 . . . . .	7
2.2	Convexification par la méthode de la plus petite valeur propre	10
2.3	Convexification par la méthode qui maximise la borne de la relaxation continue (QCR) . . . . .	11
<b>3</b>	<b>Une nouvelle méthode de convexification directe des programmes quadratiques en nombres entiers : méthode "semi 0-1"</b>	<b>14</b>
<b>4</b>	<b>Une nouvelle méthode de linéarisation des programmes quadratiques en nombres entiers</b>	<b>17</b>
<b>5</b>	<b>Résultats expérimentaux et étude comparative des différentes méthodes</b>	<b>20</b>
5.1	Résultats des test des instances IQKP(1) . . . . .	22
5.2	Résultats des test des instances IQKP(2) . . . . .	23
5.3	Résultats des test des instances UIQP(1) . . . . .	25
<b>6</b>	<b>Conclusion</b>	<b>27</b>
	<b>Références</b>	<b>28</b>
<b>7</b>	<b>Annexes</b>	<b>29</b>
<b>A</b>	<b>Suivi d'un exemple détaillé</b>	<b>29</b>
A.1	Reformulation du problème en variables 0-1 . . . . .	29
A.2	Résolution par la méthode de la plus petite valeur propre : . .	33
A.3	Résolution par la méthode QCR : . . . . .	35
A.4	Résolution par la méthode "semi 0-1" : . . . . .	37
A.5	Résolution par la linéarisation : . . . . .	38

<b>B</b>	<b>Détails des résultats</b>	<b>39</b>
B.1	Résolution des instances IQKP(1)	39
B.2	Résolution des instances IQKP(2)	43
B.3	Résolution des instances UIQP(1)	47
<b>C</b>	<b>Réalisation Pratique</b>	<b>51</b>
C.1	Détails de l'implémentation	51
C.2	Manuel de <b>Resolution</b>	54
C.2.1	Compilation	54
C.2.2	Utilisation	54
C.2.3	Format de lecture de fichier	56
C.3	Détail du code	56

# 1 Introduction

De nombreux problèmes de recherche opérationnelle peuvent se modéliser sous la forme d'un programme mathématique à valeurs entières dont la fonction objectif est quadratique, non convexe et est soumise à des contraintes linéaires. Sans perte de généralité, un problème de ce type peut s'écrire sous la forme :

$$(P) \begin{cases} \text{Min} & f(x) = x^T Q x + c^T x \\ \text{S.c.} & Ax \leq b & (1) \\ & Dx = e & (2) \\ & 0 \leq x_i \leq u_i & (3) \\ & x_i \in \mathbf{N} & (4) \end{cases}$$

Où les matrices et vecteurs sont définis comme suit :

- La matrice  $Q$  est symétrique et de dimension  $n * n$
- Le vecteur  $c$  est de dimension  $n$
- la matrice  $A$  est de dimension  $n * m$
- le vecteur  $b$  est de dimension  $m$
- la matrice  $D$  est de dimension  $n * p$
- le vecteur  $e$  est de dimension  $p$
- le vecteur  $u$ , chaque  $u_i$  correspondant à la borne supérieure de la valeur autorisée à la variable  $x_i$  et de dimension  $n$ .

Ce problème entre dans la classe des problèmes difficiles, car il doit traiter à la fois une fonction objectif non linéaire et trouver une solution à valeurs entières.

Nous allons donc proposer une série de transformations de ce problème ( $P$ ) en un problème ( $P'$ ) équivalent, dont la fonction économique est quadratique et convexe, et que nous pourrons donc résoudre grâce à un solveur comme XPress.

Pour évaluer la qualité de la transformation apportée, nous évaluerons la valeur de la relaxation continue de ( $P'$ ), qui nous fournit une borne inférieure de l'optimum entier recherché. La valeur de cette borne doit être la plus proche possible de l'optimum en question pour démarrer une procédure de Branch and Bound, mise en oeuvre lors de la résolution d'un programme mathématique en nombres entiers.

Les méthodes de convexification connues sont en général des convexifi-

cations lorsque le problème est à valeurs dans  $\{0, 1\}$ , car le fait de forcer le vecteur solution à ne prendre qu'une de ces deux valeurs nous donne une propriété qui facilite la convexification :

si  $t \in \{0, 1\}$  , alors  $t^2 - t = 0$

L'intérêt de cette propriété  $t^2 - t = 0$  est qu'elle nous permet d'ajouter un certain nombre  $\lambda$  de fois cette différence à notre fonction quadratique non convexe  $f(t)$ , jusqu'à ce que son hessien soit semi défini positif, et donc que  $f(t)$  soit convexe sans en modifier la valeur en  $\{0, 1\}$ .

Avec la propriété vue précédemment, on a :

si  $t \in \{0, 1\}$  , alors  $f(t) = f(t) + \sum_{i=1}^n \lambda_i (t_i^2 - t_i)$ .

Une façon de faire consiste à transformer notre problème à solutions entières en un problème à solutions à valeurs dans  $\{0, 1\}$  afin de lui appliquer ces méthodes. Parmi les méthodes de convexification connues, nous allons en utiliser deux la première qui est la méthode de la plus petite valeur propre [1] [2]. La deuxième est la méthode des coefficients par la programmation semi-définie : méthode dite QCR [3]. Elles ont été étudiées pour des programmes quadratiques à variables dans  $\{0, 1\}$  du type :

$$(P_{01}) \left\{ \begin{array}{ll} \text{Min} & f(t) = t^T * \hat{Q} * t + \hat{c}^T * t \\ \text{S.c} & \hat{A} * t \leq b \quad (1) \\ & \hat{D} * t = e \quad (2) \\ & t_i \in \{0, 1\} \quad (3) \end{array} \right.$$

Nous introduirons également une nouvelle convexification qui ne nécessite pas de transformation préalable des variables en  $\{0, 1\}$ . Cette méthode convexifie  $f(x)$  en utilisant le principe de la plus petite valeur propre de  $Q$  [1]. Enfin, nous comparerons l'efficacité de ces différentes méthodes de convexification avec une linéarisation. Ces deux reformulations s'appliquent directement à des problèmes de type  $(P)$ .

Dans la suite de cette étude nous expliquerons le principe de chacune de ces méthodes et tenterons de les appliquer à notre problème afin d'évaluer si une méthode de résolution existante est efficace ou s'il est nécessaire d'en trouver une nouvelle. Commençons d'abord par expliquer la transformation du problème en nombres entiers en un problème en variables 0-1.

## 2 Résolution d'un programme quadratique en nombres entiers par transformation en un programme quadratique en variables 0-1

### 2.1 Reformulation du problème en variables 0-1

Pour résoudre le problème ( $P$ ) en nombres entiers, on peut donc transformer notre programme quadratique en un programme quadratique équivalent qui ne contient que des variables prenant les valeurs 0 ou 1. Après avoir effectué cette opération, il est possible de convexifier la fonction objectif par des méthodes connues, notamment celles qui utilisent la plus petite valeur propre de  $Q$ , ou bien un vecteur  $\lambda$  plus approprié qui maximiserait la borne nécessaire au branch and bound.

Nous nous intéresserons donc dans cette section à la façon de modéliser notre programme quadratique ( $P$ ) défini précédemment en un programme équivalent ayant des solutions de valeur 0 ou 1. La façon la plus logique pour effectuer ce changement est de transformer notre vecteur solution  $x = (x_1, \dots, x_i, \dots, x_n)$  en décomposant les  $x_i$  en puissances de 2 et donc de la façon suivante :

$$x_i = \sum_{k=0}^{\lfloor \log(u_i) \rfloor} 2^k * t_{ik} \text{ où } t_{ik} \in \{0, 1\}$$

On obtient ainsi un nouveau vecteur solution :

$$t = (t_{10}, \dots, t_{1\lfloor \log(u_1) \rfloor}, t_{20}, \dots, t_{2\lfloor \log(u_2) \rfloor}, \dots, t_{n0}, \dots, t_{n\lfloor \log(u_n) \rfloor})$$

Notre programme quadratique devient donc :

$$(P') \begin{cases} \text{Min} & f(t) = t^T * \hat{Q} * t + \hat{c}^T * t \\ \text{S.c.} & \hat{A} * t \leq b \\ & \hat{D} * t = e \\ & t_{ik} \in \{0, 1\}, i = 1, \dots, n, k = 0, \dots, \lfloor \log(u_i) \rfloor \end{cases} \begin{matrix} (1) \\ (2) \\ (3) \end{matrix}$$

Si on pose  $N = \sum_{i=1}^n (\lfloor \log(u_i) \rfloor + 1)$ , les matrices et vecteurs  $\hat{Q}$ ,  $\hat{A}$ ,  $\hat{D}$  et  $\hat{c}$  sont définis de la manière suivante :

- $\hat{Q}$  est la matrice  $Q$  de dimension  $n * n$  transformée. La nouvelle matrice  $\hat{Q}$  a comme dimension  $N * N$  et est composée des coefficients suivants :



$$(\hat{Q}) \left[ \begin{array}{l} \hat{Q}_{1010}, \dots, \hat{Q}_{101[\log(u_1)]}, \dots, \hat{Q}_{10i0}, \dots, \hat{Q}_{10i[\log(u_i)]}, \dots, \hat{Q}_{10n0}, \dots, \hat{Q}_{10n[\log(u_n)]} \\ \hat{Q}_{1110}, \dots, \hat{Q}_{111[\log(u_1)]}, \dots, \hat{Q}_{11i0}, \dots, \hat{Q}_{11i[\log(u_i)]}, \dots, \hat{Q}_{11n0}, \dots, \hat{Q}_{11n[\log(u_n)]} \\ \hat{Q}_{2010}, \dots, \hat{Q}_{201[\log(u_1)]}, \dots, \hat{Q}_{20i0}, \dots, \hat{Q}_{20i[\log(u_i)]}, \dots, \hat{Q}_{20n0}, \dots, \hat{Q}_{20n[\log(u_n)]} \\ \hat{Q}_{n[\log(u_n)]10}, \dots, \hat{Q}_{n[\log(u_n)]1[\log(u_1)]}, \dots, \hat{Q}_{n[\log(u_n)]n1}, \dots, \hat{Q}_{n[\log(u_n)]n[\log(u_n)]} \end{array} \right]$$

Où le coefficient  $\hat{q}_{ikjl} = q_{ij} * 2^k * 2^l$  est celui du produit des variables  $t_{ik}$  et  $t_{jl}$ , en effet on a :

$$q_{ij} * x_i * x_j = q_{ij} * \left( \sum_{k=0}^{[\log(u_i)]} 2^k * t_{ik} \right) * \left( \sum_{l=0}^{[\log(u_j)]} 2^l * t_{jl} \right)$$

$$q_{ij} * x_i * x_j = \sum_{k=0}^{[\log(u_i)]} \sum_{l=0}^{[\log(u_j)]} (q_{ij} * 2^k * 2^l * t_{ik} * t_{jl})$$

$$q_{ij} * x_i * x_j = \sum_{k=0}^{[\log(u_i)]} \sum_{l=0}^{[\log(u_j)]} (\hat{q}_{ikjl} * t_{ik} * t_{jl})$$

$$\text{Donc } \hat{q}_{ikjl} = q_{ij} * 2^k * 2^l$$

–  $\hat{c}$  est le vecteur  $c$  de dimension  $n$  transformé, ce nouveau vecteur a comme dimension  $N$  et est composé des coefficients suivants :

$$\hat{c} = (\hat{c}_{10}, \dots, \hat{c}_{1[\log(u_1)]}, \hat{c}_{i0}, \dots, \hat{c}_{i[\log(u_i)]}, \hat{c}_{n0}, \dots, \hat{c}_{n[\log(u_n)]})$$

Où le coefficient  $\hat{c}_{ik} = c_i * 2^k$  est celui de la variable  $t_{ik}$ , en effet, on a :

$$c_i * x_i = c_i * \left( \sum_{k=0}^{[\log(u_i)]} 2^k * t_{ik} \right)$$

$$c_i * x_i = \sum_{k=0}^{[\log(u_i)]} (c_i * 2^k * t_{ik})$$

$$c_i * x_i = \sum_{k=0}^{[\log(u_i)]} (\hat{c}_{ik} * t_{ik})$$

$$\text{Donc } \hat{c}_{ik} = c_i * 2^k$$

- $\hat{A}$  est la matrice  $A$  de dimension  $n * m$  transformée. La nouvelle matrice  $\hat{A}$  a comme dimension  $N * m$  et est composée des coefficients suivants :

$$(\hat{A}) \left[ \begin{array}{l} \hat{a}_{110}, \dots, \hat{a}_{11[\log(u_1)]}, \dots, \hat{a}_{1i0}, \dots, \hat{a}_{1i[\log(u_i)]}, \dots, \hat{a}_{1n0}, \dots, \hat{a}_{1n[\log(u_n)]} \\ \hat{a}_{210}, \dots, \hat{a}_{21[\log(u_1)]}, \dots, \hat{a}_{2i0}, \dots, \hat{a}_{2i[\log(u_i)]}, \dots, \hat{a}_{2n0}, \dots, \hat{a}_{2n[\log(u_n)]} \\ \dots \\ \hat{a}_{m10}, \dots, \hat{a}_{m1[\log(u_1)]}, \dots, \hat{a}_{mi0}, \dots, \hat{a}_{mi[\log(u_i)]}, \dots, \hat{a}_{mn0}, \dots, \hat{a}_{mn[\log(u_n)]} \end{array} \right]$$

Où le coefficient  $\hat{a}_{ijl} = a_{ij} * 2^l$  est celui de la variable  $t_{jl}$  dans la contrainte  $i$ .

$$a_{ij} * x_j = a_{ij} * \left( \sum_{l=0}^{[\log(u_j)]} 2^l * t_{jl} \right)$$

$$a_{ij} * x_j = \sum_{l=0}^{[\log(u_j)]} (a_{ij} * 2^l * t_{jl})$$

$$a_{ij} * x_j = \sum_{l=0}^{[\log(u_j)]} (\hat{a}_{ijl} * t_{jl})$$

$$\text{Donc } \hat{a}_{ijl} = a_{ij} * 2^l$$

- $\hat{D}$  est la matrice  $D$  de dimension  $n * p$  transformée. La nouvelle matrice  $\hat{D}$  a comme dimension  $N * p$  et est composée des coefficients suivants :

$$(\hat{D}) \left[ \begin{array}{l} \hat{d}_{110}, \dots, \hat{d}_{11[\log(u_1)]}, \dots, \hat{d}_{1i0}, \dots, \hat{d}_{1i[\log(u_i)]}, \dots, \hat{d}_{1n0}, \dots, \hat{d}_{1n[\log(u_n)]} \\ \hat{d}_{210}, \dots, \hat{d}_{21[\log(u_1)]}, \dots, \hat{d}_{2i0}, \dots, \hat{d}_{2i[\log(u_i)]}, \dots, \hat{d}_{2n0}, \dots, \hat{d}_{2n[\log(u_n)]} \\ \dots \\ \hat{d}_{l10}, \dots, \hat{d}_{l1[\log(u_1)]}, \dots, \hat{d}_{li0}, \dots, \hat{d}_{li[\log(u_i)]}, \dots, \hat{d}_{ln0}, \dots, \hat{d}_{ln[\log(u_n)]} \end{array} \right]$$

Où le coefficient  $\hat{d}_{ijl} = d_{ij} * 2^l$  est celui de la variable  $t_{jl}$  dans la contrainte  $i$ .

$$d_{ij} * x_j = d_{ij} * \left( \sum_{l=0}^{[\log(u_j)]} 2^l * t_{jl} \right)$$

$$d_{ij} * x_j = \sum_{l=0}^{[\log(u_j)]} (d_{ij} * 2^l * t_{jl})$$

$$d_{ij} * x_j = \sum_{l=0}^{\lfloor \log(u_j) \rfloor} (\hat{d}_{ijl} * t_{jl})$$

Donc  $\hat{d}_{ijl} = d_{ij} * 2^l$

– Les vecteurs  $b$  et  $e$  sont les mêmes que dans  $(P)$ .

La complexité totale de cette transformation est polynomiale, il est intéressant d'étudier les méthodes de convexifications existantes sur les programmes quadratiques en variables 0-1 après avoir effectué cette transformation sur notre programme quadratique  $(P)$  de départ.

## 2.2 Convexification par la méthode de la plus petite valeur propre

Cette méthode a été étudiée pour le problème quadratique en variables 0-1, obtenu à la fin du paragraphe précédent suivant :

$$(P') \begin{cases} \text{Min} & f(t) = t^T * \hat{Q} * t + \hat{c}^T * t \\ \text{S.c} & \hat{A} * t \leq b \\ & \hat{D} * t = e \\ & t_{ik} \in \{0, 1\}, i = 1, \dots, n, k = 0, \dots, \lfloor \log(u_i) \rfloor \end{cases} \begin{matrix} (1) \\ (2) \\ (3) \end{matrix}$$

Comme nous travaillons sur un espace de solution qui est  $\{0, 1\}$ , nous avons donc la propriété que :  
si  $t_{ik} \in \{0, 1\}$ , alors  $t_{ik}^2 - t_{ik} = 0$ .

De plus,  $\hat{Q}$  est symétrique car la reformulation décrite précédemment, d'une matrice symétrique donne une matrice symétrique.

Ensuite, notons  $diag(\lambda)$  la matrice diagonale de dimension  $N * N$  obtenue depuis le vecteur  $\lambda$  de dimension  $N$  définit comme suit :

$$\lambda = (\lambda_{10}, \dots, \lambda_{1\lfloor \log(u_1) \rfloor}, \lambda_{i0}, \dots, \lambda_{i\lfloor \log(u_i) \rfloor}, \lambda_{n0}, \dots, \lambda_{n\lfloor \log(u_n) \rfloor})$$

$$f_\lambda(t) = t^T * (\hat{Q} - diag(\lambda)) * t + (\hat{c} + \lambda)^T * t$$

$$f_\lambda(t) = f(t) + \sum_{i=1}^n \sum_{k=0}^{\lfloor \log(u_i) \rfloor} \lambda_{ik} (t_{ik} - t_{ik}^2)$$

Il est évident que :

$$f_\lambda(t) = f(t) \quad \forall \lambda, \text{ si } t_{ik} \in \{0, 1\}^n$$

Notre programme quadratique ( $P'$ ) peut donc s'écrire en un programme quadratique ( $P_\lambda$ ) équivalent :

$$(P_\lambda) \begin{cases} \text{Min} & f_\lambda(t) = t^T * \hat{Q} * t + \hat{c}^T * t + \sum_{i=1}^n \sum_{k=0}^{\lfloor \log(u_i) \rfloor} \lambda_{ik} (t_{ik} - t_{ik}^2) \\ \text{S.c} & \hat{A} * t \leq b \\ & \hat{D} * t = e \\ & t_{ik} \in \{0, 1\}, i = 1, \dots, n, k = 0, \dots, \lfloor \log(u_i) \rfloor \end{cases} \begin{matrix} (1) \\ (2) \\ (3) \end{matrix}$$

Pour que  $f_\lambda(t)$  soit convexe, il faut trouver un vecteur  $\lambda$  tel que la matrice  $\hat{Q} - \text{diag}(\lambda)$  soit semi-définie positive.

**Propriété 2.1** Soit  $\lambda_{min}$  la plus petite valeur propre de  $\hat{Q}$ , on a la propriété que si au moins un terme de  $\hat{Q}$  est différent de 0 alors  $\lambda_{min} \leq 0$ , et la matrice  $\hat{Q} - \text{diag}(\lambda_{min} * e)$  est semi définie positive.

Posons  $\lambda = \lambda_{min} * e$  avec  $e$  le vecteur identité,  
et  $f_{\lambda_{min}*e} = t^T * (\hat{Q} - \text{diag}(\lambda_{min} * e)) * t + (\hat{c} + \lambda_{min} * e)^T * t$

**Propriété 2.2** ( $P'$ ) équivalent à ( $P_\lambda$ ), avec la matrice  $(\hat{Q} - \text{diag}(\lambda_{min} * e))$  qui est semi-définie positive et donc  $f_{\lambda_{min}*e}$  qui est convexe .

### 2.3 Convexification par la méthode qui maximise la borne de la relaxation continue (QCR)

Cette méthode est une amélioration de la méthode énoncée précédemment, en effet, l'idée est de trouver une valeur meilleure pour le vecteur  $\lambda$  qui rend quand même convexe la fonction objectif  $f_\lambda$ . Cette méthode intègre également

les contraintes d'égalité à la fonction objectif afin de la convexifier au mieux.  
Soit

$$(P') \begin{cases} \text{Min} & f(t) = t^T * \hat{Q} * t + \hat{c}^T * t \\ \text{S.c} & \hat{A} * t \leq b \\ & \hat{D} * t = e \\ & t_{ik} \in \{0, 1\}, i = 1, \dots, n, k = 0, \dots, \lfloor \log(u_i) \rfloor \end{cases} \begin{matrix} (1) \\ (2) \\ (3) \end{matrix}$$

Soit  $\alpha \in \mathbf{R}^{m*N}$  et  $\lambda \in \mathbf{R}^N$  et

$$(P_{\alpha,\lambda}) \begin{cases} \text{Min} & f_{\alpha,\lambda}(t) \\ \text{S.c} & \hat{A} * t \leq b \\ & \hat{D} * t = e \\ & t_{ik} \in \{0, 1\}, i = 1, \dots, n, k = 0, \dots, \lfloor \log(u_i) \rfloor \end{cases} \begin{matrix} (1) \\ (2) \\ (3) \end{matrix}$$

$$\text{avec } f_{\alpha,\lambda}(t) = f(t) + \sum_{v=1}^m \sum_{i=1}^N \sum_{k=0}^{\lfloor \log(u_i) \rfloor} \alpha_{vik} * t_{ik} \left( \sum_{j=1}^N \sum_{l=0}^{\lfloor \log(u_j) \rfloor} \hat{a}_{vjl} * t_{jl} - b_v \right) \\ + \sum_{i=1}^n \sum_{k=0}^{\lfloor \log(u_i) \rfloor} \lambda_{ik} (t_{ik} - t_{ik}^2)$$

$$f_{\alpha,\lambda}(t) = t^T * \hat{Q}_{\alpha,\lambda} * t + \hat{c}_{\alpha,\lambda}^T * t \text{ où}$$

$$\hat{Q}_{\alpha,\lambda} = \hat{Q} + 1/2(\alpha^T * \hat{A} + \hat{A}^T * \alpha) + \text{diag}(\lambda) \text{ et}$$

$$\hat{c}_{\alpha,\lambda} = \hat{c} - \alpha^T * b - \lambda$$

On a bien  $f(t) = f_{\alpha,\lambda}(t)$  quand  $\hat{A} * t = b$  et  $t_{ik} \in \{0, 1\}^n$   
De plus on peut avoir  $f_{\alpha,\lambda}(t)$  convexe, par exemple lorsque  $\alpha = 0$  et  $\lambda = \lambda_{min} * e$ .

On va donc déterminer  $(\alpha, \lambda)$  tel que  $f_{\alpha,\lambda}(t)$  soit convexe et que la relaxation de  $(P_{\alpha,\lambda})$  soit maximale et donc résoudre le problème suivant :

$$(C(P_{qcr})) \begin{cases} \text{Max} & \text{Min } g_{\alpha,\lambda}(t) \\ \text{S.c} & \alpha \in \mathbf{R}^{m*N} \\ & \lambda \in \mathbf{R}^N \\ & Q_{\alpha,\lambda} \geq 0 \end{cases} \begin{matrix} (1) \\ (2) \\ (3) \end{matrix}$$

**Théorème 2.1**  $(C(P_{qcr}))$  est équivalent au dual de la relaxation semie-définie  $(SDP_{qcr})$  du problème  $(P_{qcr})$ . La solution optimale  $(\alpha^*, \lambda^*)$  de  $(C(P'))$  peut être obtenue en résolvant  $(SDP_{qcr})$ .

La valeur optimale de  $\lambda^*$  est donnée par la valeur optimale du dual des variables associées aux contraintes (1) et celle de  $\alpha^*$  est donnée par la valeur optimale du dual des variables associées aux contraintes (2), avec  $f_{\alpha^*, \lambda^*}(t)$  convexe.

$$\left( SDP_{qcr} \right) \left\{ \begin{array}{l}
 \text{Min} \quad \hat{c}^T * t + \sum_{i=1}^n \sum_{k=0}^{\lfloor \log(u_i) \rfloor} \sum_{j=1}^m \sum_{l=0}^{\lfloor \log(u_j) \rfloor} \hat{Q}_{ikjl} * T_{ikjl} \\
 \text{S.c} \quad T_{ikik} = t_{ik} \quad i = 1, \dots, n, \quad k = 0, \dots, \lfloor \log(u_i) \rfloor \quad (1) \\
 -b_v * t_{ik} + \sum_{j=1}^n \sum_{l=0}^{\lfloor \log(u_j) \rfloor} \hat{a}_{vjl} * T_{ikjl} = 0 \quad v = 1, \dots, m \\
 \quad \quad \quad i = 1 \dots n, \quad k = 0, \dots, \lfloor \log(u_i) \rfloor \quad (2) \\
 \hat{A} * t = b \\
 \hat{D} * t \leq e \\
 \begin{bmatrix} 1 & t^T \\ t & T \end{bmatrix} \geq 0. \\
 t \in \mathbf{R}^N \\
 T \in \mathbf{R}^{N*N}
 \end{array} \right.$$

### 3 Une nouvelle méthode de convexification directe des programmes quadratiques en nombres entiers : méthode "semi 0-1"

L'idée de cette transformation est d'abord d'ajouter des variables supplémentaires  $v_i = x_i^2$ , puis de perturber la fonction objectif par les fonctions nulles  $\lambda_i(x_i^2 - v_i)$ . Ensuite, par un jeu de variables et contraintes linéaires supplémentaires, il faut assurer l'égalité  $v_i = x_i^2$ . Ici, nous avons choisi de

ce faire par une décomposition en puissances de 2 de  $x_i$ ,  $x_i = \sum_{k=0}^{\lfloor \log(u_i) \rfloor} 2^k * t_{ik}$ .

Il en découle l'égalité non linéaire  $v_i = x_i^2 = \sum_{k=0}^{\lfloor \log(u_i) \rfloor} 2^k * t_{ik} * x_i$ , que nous linéarisons en introduisant les variables positives  $z_{ik} = t_{ik} * x_i$  et les contraintes de linéarisation (5), (6) et (7). Nous pouvons donc réécrire le programme mathématique quadratique ( $P$ ) sous une formulation équivalente que nous appellerons ( $P_c$ ) :

$$\left. \begin{array}{l}
 \text{Min} \quad f_\lambda(x) = x^T Q x + c^T x + \sum_{i=1}^n \lambda_i (x_i^2 - v_i) \\
 \text{S.c.} \quad Ax \leq b \quad (1) \\
 \quad \quad Dx = e \quad (2) \\
 \quad \quad x_i = \sum_{k=0}^{\lfloor \log(u_i) \rfloor} 2^k * t_{ik}, i = 1, \dots, n \quad (3) \\
 \quad \quad v_i = \sum_{k=0}^{\lfloor \log(u_i) \rfloor} 2^k * z_{ik}, i = 1, \dots, n \quad (4) \\
 \quad \quad z_{ik} \leq u_i * t_{ik}, i = 1, \dots, n, k = 0, \dots, \lfloor \log(u_i) \rfloor \quad (5) \\
 \quad \quad z_{ik} \leq x_i, i = 1, \dots, n, k = 0, \dots, \lfloor \log(u_i) \rfloor \quad (6) \\
 \quad \quad z_{ik} \geq x_i - u_i(1 - t_{ik}), i = 1, \dots, n, k = 0, \dots, \lfloor \log(u_i) \rfloor \quad (7) \\
 \quad \quad 0 \leq x_i \leq u_i, i = 1, \dots, n \quad (8) \\
 \quad \quad x_i \in \mathbf{N}, i = 1, \dots, n \quad (9) \\
 \quad \quad t_{ik} \in \{0, 1\}, i = 1, \dots, n, k = 0, \dots, \lfloor \log(u_i) \rfloor \quad (10) \\
 \quad \quad z_{ik} \geq 0, i = 1, \dots, n, k = 0, \dots, \lfloor \log(u_i) \rfloor \quad (11) \\
 \quad \quad v_i \geq 0, i = 1, \dots, n, \quad (12)
 \end{array} \right\} (P_c)$$

avec  $\lambda = \lambda_{min} * e$ ,  $\lambda_{min}$  étant la plus petite valeur propre de  $Q$ , et  $e$  le vecteur identité.

**Théorème 3.1** *Les programmes (P) et (P<sub>c</sub>) sont équivalents avec  $f_\lambda(x)$  qui est convexe.*

**Preuve :**

Montrons tout d'abord que  $z_{ik} = t_{ik} * x_i$

– Si  $t_{ik} = 0$ , montrons que  $z_{ik} = 0$

On a :

– D'après la contrainte (5)  $z_{ik} \leq u_i * t_{ik}$  donc  $z_{ik} \leq 0$ .

– D'après la contrainte (6)  $z_{ik} \leq x_i$

– D'après la contrainte (7)  $z_{ik} \geq x_i - u_i * (1 - t_{ik})$  donc  $z_{ik} \geq x_i - u_i$ .

– D'après la contrainte (11)  $z_{ik} \geq 0$ .

On a donc, si  $t_{ik} = 0$ ,  $z_{ik} = 0$

– Si  $t_{ik} = 1$ , montrons que  $z_{ik} = x_i$

On a :

– D'après la contrainte (5)  $z_{ik} \leq u_i * t_{ik}$  donc  $z_{ik} \leq u_i$ .

– D'après la contrainte (6)  $z_{ik} \leq x_i$ .

– D'après la contrainte (7)  $z_{ik} \geq x_i - u_i * (1 - t_{ik})$  donc  $z_{ik} \geq x_i$ .

– D'après la contrainte (11)  $z_{ik} \geq 0$ .

Comme on a  $x_i \leq u_i$ , on a donc, si  $t_{ik} = 1$ ,  $z_{ik} = x_i$

On a donc  $z_{ik} = t_{ik} * x_i$ .

Montrons ensuite que  $v_i = x_i^2$

On a :

$$v_i = \sum_{k=0}^{\lfloor \log(u_i) \rfloor} 2^k * z_{ik}$$

$$v_i = \sum_{k=0}^{\lfloor \log(u_i) \rfloor} 2^k * t_{ik} * x_i$$



Puisque  $z_{ik} = t_{ik} * x_i$

$$v_i = x_i * \sum_{k=0}^{\lfloor \log(u_i) \rfloor} 2^k * t_{ik}$$

$$v_i = x_i^2$$

Montrons enfin que  $f_\lambda(x) = f(x)$

$$f_\lambda(x) = x^T Q x + c^T x + \sum_{i=1}^n \lambda_i (x_i^2 - v_i)$$

Comme  $v_i = x_i^2$ , on a :

$$f_\lambda(x) = x^T Q x + c^T x$$

$$f_\lambda(x) = f(x)$$

De plus, on a  $f_\lambda(x)$  qui est convexe car on ajoute à la diagonale de son hessien sa plus petite valeur propre, et par la **Propriété 2.2.1**, celui-ci devient semi-défini positif. On peut donc résoudre ce programme quadratique grâce au logiciel XPRESS.

## 4 Une nouvelle méthode de linéarisation des programmes quadratiques en nombres entiers

L'idée est ici de remplacer chaque produit de variables  $x_i * x_j$  par une nouvelle variable entière  $y_{ij}$ . Ensuite, par un jeu de variables et contraintes linéaires supplémentaires, il faut assurer l'égalité  $y_{ij} = x_i * x_j$ . Ici, nous avons choisi de ce faire par une décomposition en puissances de 2 de  $x_i$ ,

$$x_i = \sum_{k=0}^{\lfloor \log(u_i) \rfloor} 2^k * t_{ik}. \text{ Il en découle l'égalité non linéaire } y_{ij} = x_i * x_j =$$

$\sum_{k=0}^{\lfloor \log(u_i) \rfloor} 2^k * t_{ik} * x_j$ , que nous linéarisons en introduisant les variables positives  $z_{ijk} = t_{ik} * x_j$  et les contraintes de linéarisation (5), (6) et (7). Nous pouvons donc réécrire le programme mathématique quadratique ( $P$ ) sous une formulation équivalente linéaire que nous appellerons ( $P_l$ ) :

$$(P_l) \left\{ \begin{array}{ll} \text{Min} & f_l(x, y) = \sum_{i=1}^n \sum_{j=i}^n Q_{ij} * y_{ij} + \sum_{i=1}^n \sum_{j=1}^{i-1} Q_{ij} * y_{ji} + \sum_{i=1}^n c_i * x_i \\ \text{S.c.} & Ax \leq b \quad (1) \\ & Dx = e \quad (2) \\ & x_i = \sum_{k=0}^{\lfloor \log(u_i) \rfloor} 2^k * t_{ik}, i = 1, \dots, n, j = 1, \dots, n \quad (3) \\ & y_{ij} = \sum_{k=0}^{\lfloor \log(u_i) \rfloor} 2^k * z_{ijk}, i = 1, \dots, n, j = 1, \dots, n \quad (4) \\ & z_{ijk} \leq u_j * t_{ik} \quad (5) \\ & \quad i = 1, \dots, n, j = 1, \dots, n, k = 0, \dots, \lfloor \log(u_i) \rfloor \\ & z_{ijk} \leq x_j \quad (6) \\ & \quad i = 1, \dots, n, j = 1, \dots, n, k = 0, \dots, \lfloor \log(u_i) \rfloor \\ & z_{ijk} \geq x_j - u_j(1 - t_{ik}) \quad (7) \\ & \quad i = 1, \dots, n, j = 1, \dots, n, k = 0, \dots, \lfloor \log(u_i) \rfloor \\ & 0 \leq x_i \leq u_i, i = 1, \dots, n \quad (8) \\ & x_i \in \mathbf{N}, i = 1, \dots, n \quad (9) \\ & t_{ik} \in \{0, 1\}, i = 1, \dots, n, k = 0, \dots, \lfloor \log(u_i) \rfloor \quad (10) \\ & y_{ij} \geq 0, i = 1, \dots, n, j = 1, \dots, n \quad (11) \\ & z_{ijk} \geq 0, i = 1, \dots, n, k = 0, \dots, \lfloor \log(u_i) \rfloor \quad (12) \end{array} \right.$$

**Théorème 4.1** *Les programmes (P) et (P<sub>l</sub>) sont équivalents et f<sub>l</sub>(x, y) est linéaire.*

**Preuve :**

Montrons tout d'abord que  $z_{ijk} = t_{ik} * x_j$

– Si  $t_{ik} = 0$ , montrons que  $z_{ijk} = 0$

On a :

– D'après la contrainte (5)  $z_{ijk} \leq u_j * t_{ik}$  donc  $z_{ijk} \leq 0$ .

– D'après la contrainte (6)  $z_{ijk} \leq x_j$ .

– D'après la contrainte (7)  $z_{ijk} \geq x_j - u_j * (1 - t_{ik})$  donc  $z_{ijk} \geq x_j - u_j$ .

– D'après la contrainte (12)  $z_{ijk} \geq 0$ .

On a donc, si  $t_{ik} = 0, z_{ijk} = 0$

– Si  $t_{ik} = 1$ , montrons que  $z_{ijk} = x_j$

On a :

– D'après la contrainte (5)  $z_{ijk} \leq u_j * t_{ik}$  donc  $z_{ijk} \leq u_j$ .

– D'après la contrainte (6)  $z_{ijk} \leq x_j$ .

– D'après la contrainte (7)  $z_{ijk} \geq x_j - u_j * (1 - t_{ik})$  donc  $z_{ijk} \geq x_j$ .

– D'après la contrainte (12)  $z_{ijk} \geq 0$ .

Comme on a  $x_j \leq u_j$ , on a donc, si  $t_{ik} = 1, z_{ijk} = x_j$

On a donc  $z_{ijk} = t_{ik} * x_j$ .

Montrons ensuite que  $y_{ij} = x_i * x_j$

On a :

$$y_{ij} = \sum_{k=0}^{\lfloor \log(u_i) \rfloor} 2^k * z_{ijk}$$

$$y_{ij} = \sum_{k=0}^{\lfloor \log(u_i) \rfloor} 2^k * t_{ik} * x_j$$

Puisque  $z_{ijk} = t_{ik} * x_j$

$$y_{ij} = x_j * \sum_{k=0}^{\lfloor \log(u_i) \rfloor} 2^k * t_{ik}$$

$$y_{ij} = x_i * x_j$$

Montrons enfin que  $f_l(x, y) = f(x)$

$$f_l(x, y) = \sum_{i=1}^n \sum_{j=i}^n Q_{ij} * y_{ij} + \sum_{i=1}^n \sum_{j=1}^{i-1} Q_{ij} * y_{ji} + \sum_{i=1}^n c_i * x_i$$

Comme  $y_{ij} = x_i * x_j$ , on a :

$$f_l(x, y) = \sum_{i=1}^n \sum_{j=i}^n Q_{ij} * x_i * x_j + \sum_{i=1}^n \sum_{j=1}^{i-1} Q_{ij} * x_j * x_i + \sum_{i=1}^n c_i * x_i$$

$$f_l(x, y) = \sum_{i=1}^n \sum_{j=1}^n Q_{ij} * x_i * x_j + \sum_{i=1}^n c_i * x_i$$

$$f_l(x, y) = f(x)$$

De plus, on a  $f_l(x, y)$  qui est linéaire, on peut donc résoudre ce programme linéaire grâce au logiciel XPRESS.

## 5 Résultats expérimentaux et étude comparative des différentes méthodes

Les tests sur l'efficacité des différentes méthodes ont été réalisés en partie sur les instances du problème du sac à dos quadratique suivantes :

$$(IQKP) \begin{cases} \text{Min} & f(x) = x^T * Q * x + c^T * x \\ \text{S.c} & a^T * x \leq b \\ & 0 \leq x_i \leq u_i \quad i = 1, \dots, n \end{cases}$$

Et sur le problème quadratique sans contraintes suivant :

$$(UIQP) \begin{cases} \text{Min} & f(x) = x^T * Q * x + c^T * x \\ & 0 \leq x_i \leq u_i \quad i = 1, \dots, n \end{cases}$$

Avec  $Q$  une matrice symétrique de taille  $n * n$ ,  $a$ ,  $c$  et  $u$  des vecteurs de taille  $n$ . Toutes les valeurs numériques de ces instances sont entières, et pour le vecteur  $a$  elles sont en plus positives.

Nous avons défini deux variantes de ces instances qui sont les suivantes :

- **Deux catégories d'instances IQKP ( Integer Quadratic Knapsack Problem**
  - Une (IQKP(1)) où les éléments de la matrice  $Q$  et du vecteur  $c$  sont choisis aléatoirement dans l'intervalle  $[-40, 20]$ , avec  $diag(Q) = 0$ . Les éléments du vecteur  $a$  sont choisis dans l'intervalle  $[1, 40]$ , la valeur de  $b$  est  $\sum_{i=1}^n a_i$ , et chaque  $u_i$  vaut  $\lfloor b/a_i \rfloor$ .
  - L'autre (IQKP(2)) où les éléments de la matrice  $Q$  et du vecteur  $c$  sont choisis aléatoirement dans l'intervalle  $[-100, 100]$ . Les éléments du vecteur  $a$  sont choisis dans l'intervalle  $[1, 50]$ , la valeur de  $b$  est  $20 * \sum_{i=1}^n a_i$ , et chaque  $u_i$  vaut 50.
- **une catégorie d'instances UIQP (Unconstrained Integer Quadratic Problem)**, où les éléments de la matrice  $Q$  et du vecteur  $c$  sont choisis aléatoirement dans l'intervalle  $[-100, 100]$ , et chaque  $u_i$  vaut 50.

Les statistiques évaluent les critères suivants :

- **nb\_var** : qui est le nombre réel de variables de l'instance, c'est à dire soit après une reformulation en 0-1, soit avec les changements de variables imposés par la convexification "semi 0-1" et la linéarisation.
- **gap** : qui est égal au rapport suivant :

$$\frac{\text{valeur de la relaxation continue} - \text{valeur de la meilleure solution entiere trouvee}}{\text{valeur de la meilleure solution entiere trouvee}}$$

- **noeuds** : qui est le nombre de noeuds parcouru lors du Branch and Bound pour trouver la meilleure solution entière.
- **temps** : qui est le temps nécessaire à la résolution de l'instance. Comme ce temps est limité à 3600 secondes, lorsqu'il vaut 3600 secondes c'est que l'optimum entier n'a pas été trouvé.

Ces tests ont été réalisés sur une machine linux basé sur un processeur Intel Xéon, cadencé à 2.8 GHz, et résolu avec le solveur XPress-Mosel version 1.6.1 datant de 2005.

Nous savons déjà par la théorie que le gap de la méthode de la plus petite valeur propre est toujours supérieur ou égal au gap de la méthode QCR, nous allons pouvoir comparer ces gaps avec ceux de la méthode "semi 0-1" et de la linéarisation.

Pour chaque valeur différente de  $n \in \{5, 10, 15, 20, 25\}$ , une moyenne sur 5 instances résolues par chaque méthode est résumée dans les tableaux suivant :

## 5.1 Résultats des test des instances IQKP(1)

n	nb_var	gap(%)	noeuds	temps(s)
5	16	316	194	0
10	40	1073	175000	906
15	70	1602	2834000	3600
20	101	2981	1910000	3600
25	138	3712	980000	3600

FIG. 1 – Résultats IQKP(1) de la résolution par la méthode de la valeur propre

n	nb_var	gap(%)	noeuds	temps(s)
5	16	165	55	0
10	40	305	10155	3
15	70	277	528250	736
20	101	433	1608000	2633
25	138	423	1380000	3600

FIG. 2 – Résultats IQKP(1) de la résolution par la méthode QCR

n	nb_var	gap(%)	noeuds	temps(s)
5	26	626	417	0
10	60	660	62835	128
15	100	725	1536000	3600
20	141	1878	1130000	3600
25	188	2065	596	3600

FIG. 3 – Résultats IQKP(1) de la résolution par la convexification "semi 0-1"

n	nb_var	gap(%)	noeuds	temps(s)
5	79	100	16	0
10	331	221	219	0,4
15	767	206	516	2
20	1382	264	3767	27
25	2109	233	645	15

FIG. 4 – Résultats IQKP(1) de la résolution par la linéarisation

Les résultats de ces tests qui sont effectués sur des instances où les contraintes sont très serrées montrent une nette efficacité de la linéarisation qui fournit à la fois les meilleures bornes et un temps de résolution le plus court. Nous pouvons également noter que les bornes fournies par la méthode QCR sont nettement meilleures que celles des autres convexifications, cela est sans doute dû au fait que cette méthode choisit bien plus finement son vecteur de convexification que les deux autres méthodes.

## 5.2 Résultats des test des instances IQKP(2)

n	nb_var	gap(%)	noeuds	temps(s)
5	30	301	998504	1989
10	60	202	1620000	3600
15	90	219	840000	3600
20	120	266	488000	3600
25	150	219	304000	3600

FIG. 5 – Résultats IQKP(2) de la résolution par la méthode de la valeur propre



n	nb_var	gap(%)	noeuds	temps(s)
5	30	60	445	0
10	60	56	34000	72
15	90	48	521600	1562
20	120	72	326000	3600
25	150	50	232000	3600

FIG. 6 – Résultats IQKP(2) de la résolution par la méthode QCR

n	nb_var	gap(%)	noeuds	temps(s)
5	40	48	10584	1
10	80	27	168600	795
15	120	18	325414	2002
20	160	31	287800	3600
25	200	22	274000	3600

FIG. 7 – Résultats IQKP(2) de la résolution par la convexification "semi 0-1"

n	nb_var	gap(%)	noeuds	temps(s)
5	140	57	74	0
10	455	52	577	4
15	945	67	2110	85
20	1610	119	26800	3600
25	2450	102	11600	3600

FIG. 8 – Résultats IQKP(2) de la résolution par la linéarisation

Lorsque les contraintes sont plus lâches, les performances des différentes méthodes sont bien différentes. En effet, c'est la méthode de convexification "semi 0-1" qui obtient les bornes les plus performantes même si elle résout moins d'instances en 1 heure que la linéarisation. De plus, la méthode QCR obtient également des bornes intéressantes qui sont quatre à six fois meilleures que celle de la méthode de la valeur propre. Ainsi en adaptant la méthode QCR à la convexification "semi 0-1", on pourrait certainement obtenir des bornes et temps de calculs encore meilleurs. Notons également que lors de la résolution des instances IQKP(1), le gap de la méthode QCR était toujours meilleur que celui de la méthode "semi 0-1", alors que c'est l'inverse pour la résolution des instances IQKP(2). Nous ne pourrions donc pas montrer que

le gap d'une de ces deux méthodes est toujours meilleur que celui de l'autre.

### 5.3 Résultats des test des instances UIQP(1)

n	nb_var	gap(%)	noeuds	temps(s)
5	30	111	277800	324
10	60	182	1560000	3600
15	90	212	976000	3600
20	120	199	558000	3600
25	150	221	328000	3600

FIG. 9 – Résultats UIQP(1) de la résolution par la méthode de la valeur propre

n	nb_var	gap(%)	noeuds	temps(s)
5	30	37	445	0
10	60	52	369047	760
15	90	61	838000	3600
20	120	44	466000	3600
25	150	51	456000	3600

FIG. 10 – Résultats UIQP(1) de la résolution par la méthode QCR

n	nb_var	gap(%)	noeuds	temps(s)
5	40	5	194	0,4
10	80	14	15645	83
15	120	17	332000	2827
20	160	26	368000	3600
25	200	20	348000	3600

FIG. 11 – Résultats UIQP(1) de la résolution par la convexification "semi 0-1"

n	nb_var	gap(%)	noeuds	temps(s)
5	140	16	14	0
10	455	70	819	10
15	945	106	5557	289
20	1610	98	15613	2726
25	2450	133	9267	3600

FIG. 12 – Résultats UIQP(1) de la résolution par la linéarisation

Les résultats de ces tests sans contraintes sont encore résolus le plus rapidement par la linéarisation, mais la convexification "semi 0-1" donne des bornes très prometteuses qui sont les meilleures sur ces instances. Ajoutons qu'il semble normal que la méthode QCR ne donne pas de résultats significatifs, puisque sans contraintes elle ne peut pas affiner la recherche de son vecteur de convexification.

## 6 Conclusion

Cette étude a permis d'évaluer la qualité de plusieurs méthodes de résolution de programmes quadratiques non convexes soumis à des contraintes linéaires et à valeurs entières. Nous nous sommes plus particulièrement intéressés à des méthodes de convexification. Tout d'abord, en adaptant les méthodes de la plus petite valeur propre et QCR, conçues pour des programmes quadratiques non convexes à valeurs dans  $\{0, 1\}$  et soumis à des contraintes linéaires. Puis, en convexifiant notre problème sans transformation préalable par la méthode "semi 0-1". Enfin, nous avons comparé la qualité de ces différentes résolutions avec celle d'une linéarisation.

Les méthodes impliquant une transformation préalable ne sont pas efficaces sur les instances que nous avons testées. Cela n'est pas étonnant car la transformation appliquée à notre problème augmente de façon significative son nombre de variables. Par contre, même si la linéarisation reste la plus rapide à la résolution et résout également le plus grand nombre d'instances en 1 heure, dans de nombreux cas, sa relaxation continue est plus mauvaise que celle de la convexification "semi 0-1". Cela peut s'expliquer par le fait que la convexification "semi 0-1" est en fait une linéarisation des termes au carré de la fonction objectif, cette transformation a donc moins de variables que la linéarisation, ce qui n'est pas négligeable pour des problèmes de très grandes tailles.

Ainsi, la méthode de convexification "semi 0-1" semble prometteuse, car elle utilise actuellement la méthode de la plus petite valeur propre pour convexifier la fonction objectif, mais en lui adaptant la méthode QCR ou en trouvant un vecteur de convexification plus adapté, grâce à l'utilisation du lagrangien [4] [5], ou à des méthodes de programmation semi-définie [6], elle aurait une relaxation continue encore meilleure et donc sûrement un temps de résolution réduit.

De plus, on pourrait également imaginer de convexifier notre problème directement en nombre entiers en ajoutant un nombre minimum de variables, soit utilisant des méthodes de programmation semi-définie [6] ou bien la notion de "filled fonction" [7] [8], ou alors la méthode de "la puissance  $p^{ieme}$  du lagrangien" [9]. Si cela est possible cette méthode pourrait non seulement convexifier des programmes non convexes, mais aussi "reconvexifier" des programmes déjà convexes pour leur permettre d'obtenir une meilleure relaxation continue.

## Références

- [1] P.L. Hammer et A.A. Rubin *Some remarks on quadratic programming with 0-1 variables*, **Revue Française d'Informatique et de Recherche Operationnelle**, 4(3) : 67-79, 1970.
- [2] A. Billionnet et S. Elloumi, *Using a Mixed Integer Quadratic Programming Solver for Unconstrained Quadratic 0-1 Problem*, **Mathematical Programming** ,16(2) : 188-197, 2003.
- [3] [BEP05b] A. Billionnet, S. Elloumi et M.-C. Plateau, *Convex Quadratic Programming for Exact Solution of 0-1 Quadratic Programs*, **Rapport scientifique CEDRIC**. ref. CEDRIC 856, 2005
- [4] C. Lemarechal *The omnipresence of Lagrange*, **OR**, 1 : 7-25, 2003.
- [5] A.M. Geoffrion *Lagrangian relaxation for integer programming*, **Mathematical Programming**, 2 : 82-114, 1974.
- [6] C. Lemarechal, F. Oustry *SDP relaxations in combinatorial optimization from a Lagrangian point of view*, **N. Hadjisavvas et P.M Pardalos, editeurs, Advances in Convex Analysis and Global Optimization**, **Kluwer**, pages 119-134, 2001.
- [7] Y. Shang, L. Zhang, Y. Liang *A single parameter filled function theory and algorithm for non linear integer programming*, **Journal of donghua university**, vol 22, pages 1-4, 2005.
- [8] Y. Shang, L. Zhang, *A filled function method for finding a global minimizer on global integer optimization*, **Journal of computational and Applied Mathematics**, 181 : 200-210, 2004.
- [9] D. Li et D.J. White *pt<sup>th</sup> Power Lagrangian Method for Integer Programming*, **Annals of Operations Research**, 98 : 151-170, 2000.
- [10] Inria, *Scilab 4.0 O line help*, <http://www.scilab.org/>, 2004
- [11] C. Helmberg. *A c++ implementation of the spectral bundle method*, **Manual version 1.1.1**, <http://www-user.tu-chemnitz.de/helmberg/semidef.html>, 2000.
- [12] Dash Optimization *Xpress-Mosel version 1.6.1 Xpress-Mosel language Reference Manual 1.4*, 2004, <http://www.dashoptimization.com/>, 2005.

## 7 Annexes

### A Suivi d'un exemple détaillé

Un exemple détaillé a été traité par les différentes méthodes de résolution, afin de pouvoir les comparer plus précisément. Cet exemple est une instance du sac à dos quadratique.

Soit le programme linéaire de 5 variables en nombres entiers suivant :

$$(QK) \left\{ \begin{array}{l} \text{Min} \quad f(x) = -32x_1x_2 - 4x_1x_3 - 32x_1x_4 + 10x_1x_5 - 26x_2x_3 \\ \quad \quad -16x_2x_4 - 28x_2x_5 + 8x_3x_4 + 6x_3x_5 - 2x_4x_5 \\ \quad \quad +13x_1 - 34x_2 + 3x_3 - 24x_4 - 30x_5 \\ \text{S.c} \quad 10x_1 + 7x_2 + 35x_3 + 3x_4 + 15x_5 \leq 70 \\ \quad \quad 0 \leq x_1 \leq 7 \\ \quad \quad 0 \leq x_2 \leq 10 \\ \quad \quad 0 \leq x_3 \leq 2 \\ \quad \quad 0 \leq x_4 \leq 23 \\ \quad \quad 0 \leq x_5 \leq 4 \end{array} \right.$$

On a donc les données suivantes :

$$(q) \begin{bmatrix} 0 & -16 & -2 & -16 & 5 \\ -16 & 0 & -13 & -8 & -14 \\ -2 & -13 & 0 & 4 & 3 \\ -16 & -8 & 4 & 0 & -1 \\ 5 & -14 & 3 & -1 & 0 \end{bmatrix}$$

$$c = (13, -34, 3, -24, -30), u = (7, 10, 2, 23, 4), B = 70, a = (10, 7, 35, 3, 15)$$

#### A.1 Reformulation du problème en variables 0-1

Si on lui applique la transformation en 0-1 étudiée précédemment, le changement de variables des  $x_i$  ou  $1 \leq i \leq 5$  est le suivant :

- On a  $\lfloor \log_2(u_1) \rfloor = 2$ , donc  $x_1 = t_{10} + 2t_{11} + 4t_{12}$  avec  $0 \leq t_{1j} \leq 1$  et  $0 \leq j \leq 2$
- On a  $\lfloor \log_2(u_2) \rfloor = 3$ , donc  $x_2 = t_{20} + 2t_{21} + 4t_{22} + 8t_{23}$  avec  $0 \leq t_{2j} \leq 1$  et  $0 \leq j \leq 3$
- On a  $\lfloor \log_2(u_3) \rfloor = 1$ , donc  $x_3 = t_{30} + 2t_{31}$  avec  $0 \leq t_{3j} \leq 1$  et  $0 \leq j \leq 1$

- On a  $\lfloor \log_2(u_4) \rfloor = 5$ , donc  $x_4 = t_{40} + 2t_{41} + 4t_{42} + 8t_{43} + 16t_{44}$  avec  $0 \leq t_{4j} \leq 1$  et  $0 \leq j \leq 5$
- On a  $\lfloor \log_2(u_5) \rfloor = 2$ , donc  $x_5 = t_{50} + 2t_{51} + 4t_{52}$  avec  $0 \leq t_{5j} \leq 1$  et  $0 \leq j \leq 2$

*Transformation de la fonction économique :*

$$f(x) = -32x_1x_2 - 4x_1x_3 - 32x_1x_4 + 10x_1x_5 - 26x_2x_3 - 16x_2x_4 - 28x_2x_5 + 8x_3x_4 + 6x_3x_5 - 2x_4x_5 + 13x_1 - 34x_2 + 3x_3 - 24x_4 - 30x_5$$

Effectuons le changement de variable :

$$f(x) = -32(t_{10} + 2t_{11} + 4t_{12})(t_{20} + 2t_{21} + 4t_{22} + 8t_{23}) - 4(t_{10} + 2t_{11} + 4t_{12})(t_{30} + 2t_{31}) - 32(t_{10} + 2t_{11} + 4t_{12})(t_{40} + 2t_{41} + 4t_{42} + 8t_{43} + 16t_{44}) + 10(t_{10} + 2t_{11} + 4t_{12})(t_{50} + 2t_{51}) - 26(t_{20} + 2t_{21} + 4t_{22} + 8t_{23})(t_{30} + 2t_{31}) - 16(t_{20} + 2t_{21} + 4t_{22} + 8t_{23})(t_{40} + 2t_{41} + 4t_{42} + 8t_{43} + 16t_{44}) - 28(t_{20} + 2t_{21} + 4t_{22} + 8t_{23})(t_{50} + 2t_{51}) + 8(t_{30} + 2t_{31})(t_{40} + 2t_{41} + 4t_{42} + 8t_{43} + 16t_{44}) + 6(t_{30} + 2t_{31})(t_{50} + 2t_{51}) - 2(t_{40} + 2t_{41} + 4t_{42} + 8t_{43} + 16t_{44})(t_{50} + 2t_{51}) + 13(t_{10} + 2t_{11} + 4t_{12}) - 34(t_{20} + 2t_{21} + 4t_{22} + 8t_{23}) + 3(t_{30} + 2t_{31}) - 24(t_{40} + 2t_{41} + 4t_{42} + 8t_{43} + 16t_{44}) - 30(t_{50} + 2t_{51} + 4t_{52})$$

On a donc :

$$f(t) = -32t_{10}t_{20} - 64t_{10}t_{21} - 128t_{10}t_{22} - 256t_{10}t_{23} - 4t_{10}t_{30} - 8t_{10}t_{31} - 32t_{10}t_{40} - 64t_{10}t_{41} - 128t_{10}t_{42} - 256t_{10}t_{43} - 512t_{10}t_{44} + 10t_{10}t_{50} + 20t_{10}t_{51} + 40t_{10}t_{52} - 64t_{11}t_{20} - 128t_{11}t_{21} - 256t_{11}t_{22} - 512t_{11}t_{23} - 8t_{11}t_{30} - 16t_{11}t_{31} - 64t_{11}t_{40} - 128t_{11}t_{41} - 256t_{11}t_{42} - 512t_{11}t_{43} - 1024t_{11}t_{44} + 20t_{11}t_{50} + 40t_{11}t_{51} + 80t_{11}t_{52} - 128t_{12}t_{20} - 256t_{12}t_{21} - 512t_{12}t_{22} - 1024t_{12}t_{23} - 16t_{12}t_{30} - 32t_{12}t_{31} - 128t_{12}t_{40} - 256t_{12}t_{41} - 512t_{12}t_{42} - 1024t_{12}t_{43} - 2048t_{12}t_{44} + 40t_{12}t_{50} + 80t_{12}t_{51} + 160t_{12}t_{52} - 26t_{20}t_{30} - 52t_{20}t_{31} - 16t_{20}t_{40} - 32t_{20}t_{41} - 64t_{20}t_{42} - 128t_{20}t_{43} - 256t_{20}t_{44} - 28t_{20}t_{50} - 56t_{20}t_{51} - 104t_{20}t_{52} - 52t_{21}t_{30} - 104t_{21}t_{31} - 32t_{21}t_{40} - 64t_{21}t_{41} - 128t_{21}t_{42} - 256t_{21}t_{43} - 512t_{21}t_{44} - 56t_{21}t_{50} - 112t_{21}t_{51} - 124t_{21}t_{52} - 104t_{22}t_{30} - 208t_{22}t_{31} - 64t_{22}t_{40} - 128t_{22}t_{41} - 256t_{22}t_{42} - 512t_{22}t_{43} - 1024t_{22}t_{44} - 112t_{22}t_{50} - 224t_{22}t_{51} - 448t_{22}t_{52} - 208t_{23}t_{30} - 416t_{23}t_{31} - 128t_{23}t_{40} - 256t_{23}t_{41} - 512t_{23}t_{42} - 1024t_{23}t_{43} - 2048t_{23}t_{44} - 224t_{23}t_{50} - 448t_{23}t_{51} - 996t_{22}t_{52} + 8t_{30}t_{40} + 16t_{30}t_{41} + 32t_{30}t_{42} + 64t_{30}t_{43} + 128t_{30}t_{44} + 6t_{30}t_{50} + 12t_{30}t_{51} + 24t_{30}t_{52} + 16t_{31}t_{40} + 32t_{31}t_{41} + 64t_{31}t_{42} + 128t_{31}t_{43} + 256t_{31}t_{44} + 12t_{31}t_{50} + 24t_{31}t_{51} + 48t_{31}t_{52} - 2t_{40}t_{50} - 4t_{40}t_{51} - 8t_{40}t_{52} - 4t_{41}t_{50} - 8t_{41}t_{51} - 16t_{41}t_{52} - 8t_{42}t_{50} - 16t_{42}t_{51} - 32t_{42}t_{52} - 16t_{43}t_{50} - 32t_{43}t_{51} - 64t_{43}t_{52} - 32t_{44}t_{50} - 64t_{44}t_{51} - 128t_{44}t_{52} + 13t_{10} + 26t_{11} + 52t_{12} - 34t_{20} - 68t_{21} - 136t_{22} - 272t_{23} + 3t_{30} + 6t_{31} - 24t_{40} - 48t_{41} - 96t_{42} - 192t_{43} - 384t_{44} - 30t_{50} - 60t_{51} - 120t_{52}$$

*Transformation des contraintes :*

$$10x_1 + 7x_2 + 35x_3 + 3x_4 + 15x_5 \leq 70$$

Effectuons le changement de variable :

$$10(t_{10} + 2t_{11} + 4t_{12}) + 7(t_{20} + 2t_{21} + 4t_{22} + 8t_{23}) + 35(t_{30} + 2t_{31}) + 3(t_{40} + 2t_{41} + 4t_{42} + 8t_{43} + 16t_{44}) + 15(t_{50} + 2t_{51} + 4t_{52}) \leq 70$$

On a donc

$$10t_{10} + 20t_{11} + 40t_{12} + 7t_{20} + 14t_{21} + 28t_{22} + 56t_{23} + 35t_{30} + 70t_{31} + 3t_{40} + 6t_{41} + 12t_{42} + 24t_{43} + 48t_{44} + 15t_{50} + 30t_{51} + 60t_{52} \leq 70$$

On obtient donc le programme mathématique quadratique suivant :



$$\begin{array}{l}
(QK\ 0-1) \left\{ \begin{array}{l}
Min \quad f(t) = -32t_{10}t_{20} - 64t_{10}t_{21} - 128t_{10}t_{22} - 256t_{10}t_{23} - 4t_{10}t_{30} \\
-8t_{10}t_{31} - 32t_{10}t_{40} - 64t_{10}t_{41} - 128t_{10}t_{42} - 256t_{10}t_{43} \\
-512t_{10}t_{44} + 10t_{10}t_{50} + 20t_{10}t_{51} + 40t_{10}t_{52} - 64t_{11}t_{20} \\
-128t_{11}t_{21} - 256t_{11}t_{22} - 512t_{11}t_{23} - 8t_{11}t_{30} - 16t_{11}t_{31} \\
-64t_{11}t_{40} - 128t_{11}t_{41} - 256t_{11}t_{42} - 512t_{11}t_{43} - 1024t_{11}t_{44} \\
+20t_{11}t_{50} + 40t_{11}t_{51} + 80t_{11}t_{52} - 128t_{12}t_{20} - 256t_{12}t_{21} \\
-512t_{12}t_{22} - 1024t_{12}t_{23} - 16t_{12}t_{30} - 32t_{12}t_{31} - 128t_{12}t_{40} \\
-256t_{12}t_{41} - 512t_{12}t_{42} - 1024t_{12}t_{43} - 2048t_{12}t_{44} + 40t_{12}t_{50} \\
+80t_{12}t_{51} + 160t_{12}t_{52} - 26t_{20}t_{30} - 52t_{20}t_{31} - 16t_{20}t_{40} \\
-32t_{20}t_{41} - 64t_{20}t_{42} - 128t_{20}t_{43} - 256t_{20}t_{44} - 28t_{20}t_{50} \\
-56t_{20}t_{51} - 104t_{20}t_{52} - 52t_{21}t_{30} - 104t_{21}t_{31} - 32t_{21}t_{40} \\
-64t_{21}t_{41} - 128t_{21}t_{42} - 256t_{21}t_{43} - 512t_{21}t_{44} - 56t_{21}t_{50} \\
-112t_{21}t_{51} - 124t_{21}t_{52} - 104t_{22}t_{30} - 208t_{22}t_{31} - 64t_{22}t_{40} \\
-128t_{22}t_{41} - 256t_{22}t_{42} - 512t_{22}t_{43} - 1024t_{22}t_{44} - 112t_{22}t_{50} \\
-224t_{22}t_{51} - 448t_{22}t_{52} - 208t_{23}t_{30} - 416t_{23}t_{31} - 128t_{23}t_{40} \\
-256t_{23}t_{41} - 512t_{23}t_{42} - 1024t_{23}t_{43} - 2048t_{23}t_{44} - 224t_{23}t_{50} \\
-448t_{23}t_{51} - 996t_{22}t_{52} + 8t_{30}t_{40} + 16t_{30}t_{41} + 32t_{30}t_{42} \\
+64t_{30}t_{43} + 128t_{30}t_{44} + 6t_{30}t_{50} + 12t_{30}t_{51} + 24t_{30}t_{52} \\
+16t_{31}t_{40} + 32t_{31}t_{41} + 64t_{31}t_{42} + 128t_{31}t_{43} + 256t_{31}t_{44} \\
+12t_{31}t_{50} + 24t_{31}t_{51} + 48t_{31}t_{52} - 2t_{40}t_{50} - 4t_{40}t_{51} \\
-8t_{40}t_{52} - 4t_{41}t_{50} - 8t_{41}t_{51} - 16t_{41}t_{52} - 8t_{42}t_{50} \\
-16t_{42}t_{51} - 32t_{42}t_{52} - 16t_{43}t_{50} - 32t_{43}t_{51} - 64t_{43}t_{52} \\
-32t_{44}t_{50} - 64t_{44}t_{51} - 128t_{44}t_{52} + 13t_{10} + 26t_{11} + 52t_{12} - 34t_{20} \\
-68t_{21} - 136t_{22} - 272t_{23} + 3t_{30} + 6t_{31} - 24t_{40} - 48t_{41} \\
-96t_{42} - 192t_{43} - 384t_{44} - 30t_{50} - 60t_{51} - 120t_{52} \\
S.c \quad 10t_{10} + 20t_{11} + 40t_{12} + 7t_{20} + 14t_{21} + 28t_{22} + 56t_{23} \\
+35t_{30} + 70t_{31} + 3t_{40} + 6t_{41} + 12t_{42} + 24t_{43} + 48t_{44} \\
+15t_{50} + 30t_{51} + 60t_{52} \leq 70 \\
0 \leq t_{ij}x_1 \leq 1 \text{ pour tout } i, j
\end{array} \right.
\end{array}$$

qui est constitué des données suivantes :

$$A = (10, 20, 40, 7, 14, 28, 56, 35, 70, 3, 6, 12, 24, 48, 15, 30, 60)$$

$$C = (13, 26, 52, -34, -68, -136, -272, 3, 6, -24, -48, -96, -192, -384, -30, -60, -120)$$

$$(Q) \begin{bmatrix} 0 & 0 & 0 & -16 & -32 & -64 & -128 & -2 & -4 & -16 & -32 & -64 & -128 & -256 & 5 & 10 & 20 \\ 0 & 0 & 0 & -32 & -64 & -128 & -256 & -4 & -8 & -32 & -64 & -128 & -256 & -512 & 10 & 20 & 40 \\ 0 & 0 & 0 & -128 & -256 & -512 & -8 & -16 & -32 & -64 & -128 & -256 & -512 & -1024 & 20 & 40 & 80 \\ -16 & -32 & -64 & 0 & 0 & 0 & 0 & -13 & -26 & -8 & -16 & -32 & -64 & -128 & -14 & -28 & -56 \\ -32 & -64 & -128 & 0 & 0 & 0 & 0 & -26 & -52 & -16 & -32 & -64 & -128 & -256 & -28 & -56 & -112 \\ -64 & -128 & -256 & 0 & 0 & 0 & 0 & -52 & -104 & -32 & -64 & -128 & -256 & -512 & -56 & -112 & -224 \\ -128 & -256 & -512 & 0 & 0 & 0 & 0 & -104 & -208 & -64 & -128 & -256 & -512 & -1024 & -112 & -224 & -448 \\ -2 & -4 & -8 & -13 & -26 & -52 & -104 & 0 & 0 & 4 & 8 & 16 & 32 & 64 & 3 & 6 & 12 \\ -4 & -8 & -16 & -26 & -52 & -104 & 0 & 0 & 8 & 16 & 32 & 64 & 128 & 6 & 12 & 24 \\ -16 & -32 & -64 & -8 & -16 & -32 & -64 & 4 & 8 & 0 & 0 & 0 & 0 & 0 & -1 & -2 & -4 \\ -32 & -64 & -128 & -16 & -32 & -64 & -128 & 8 & 16 & 0 & 0 & 0 & 0 & 0 & -2 & -4 & -8 \\ -64 & -128 & -256 & -32 & -64 & -128 & -256 & 16 & 32 & 0 & 0 & 0 & 0 & 0 & -4 & -8 & -16 \\ -128 & -256 & -512 & -64 & -128 & -256 & -512 & 32 & 64 & 0 & 0 & 0 & 0 & 0 & -8 & -16 & -32 \\ -256 & -512 & -1024 & -128 & -256 & -512 & -1024 & 64 & 128 & 0 & 0 & 0 & 0 & 0 & -16 & -32 & -64 \\ 5 & 10 & 20 & -14 & -28 & -56 & -112 & 3 & 6 & -1 & -2 & -4 & -8 & -16 & 0 & 0 & 0 \\ 10 & 20 & 40 & -28 & -56 & -112 & -224 & 6 & 12 & -2 & -4 & -8 & -16 & -32 & 0 & 0 & 0 \\ 20 & 40 & 80 & -56 & -112 & -224 & -448 & 12 & 24 & -4 & -8 & -16 & -32 & -64 & 0 & 0 & 0 \end{bmatrix}$$

## A.2 Résolution par la méthode de la plus petite valeur propre :

Les valeurs du vecteur propre de la matrice  $Q$  sont les suivantes :  $spectre = (-2334.26, -253.65, -39.39, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 952.14, 1675.17)$ . La valeur minimum de ce spectre est :  $-2334.26$ , le fait d'ajouter cette valeur multipliée par la différence  $(t^2 - t)$  à la fonction économique la rend convexe. Notre problème peut être résolu par le solveur XPRESS, et on obtient les résultats suivants :

*Etude de la solution continue*

La solution continue est  $(0.41, 0.34, 0.18, 0.44, 0.39, 0.27, 0.05, 0, 0, 0.48, 0.46, 0.42, 0.34, 0.18, 0.29, 0.08, 0)$  et la valeur de la fonction économique et donc de la borne est :  $-7759.861542$ , celle ci a été trouvée en 0 secondes

Les statistiques de la résolution continue sont les suivantes :

<i>Its</i>	<i>Primalobj.</i>	<i>Dualobj.</i>
0	4.0802067e + 06	- 4.3899298e + 06
1	9.5004348e + 05	- 1.0976870e + 06
2	- 6.7382672e + 03	- 1.4856349e + 04
3	- 7.4828509e + 03	- 8.8548977e + 03
4	- 7.7266560e + 03	- 7.8947524e + 03
5	- 7.7583048e + 03	- 7.7699241e + 03
6	- 7.7598458e + 03	- 7.7603852e + 03
7	- 7.7598615e + 03	- 7.7598623e + 03

*Etude de la solution en 0-1*

La solution en 0-1 est (0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0) et la valeur de la fonction économique est -1610, celle ci a été trouvée en 0 *secondes*.

Les statistiques de la résolution entière sont les suivantes :

<i>Node</i>	<i>BestSoln</i>	<i>BestBound</i>	<i>Gap</i>
10	6.000000	- 7759.861270	7765.86
20	- 547.000000	- 7759.861270	92.95%
35	- 915.000000	- 7759.861328	88.21%
37	- 1030.00000	- 7759.861328	86.73%
54	- 1359.00000	- 7455.349121	81.77%
84	- 1587.00000	- 6630.793457	76.07%
492	- 1610.0000	- 3305.325928	51.29%

Le nombre de noeuds est 873 et XPRESS a trouvé 7 solutions en 0-1.

*Reconstruction de la solution :*

On avait décomposé nos variables  $x_i$  de la façon suivante :

- $x_1 = t_{10} + 2t_{11} + 4t_{12}$ , donc  $x_1 = 1 * 0 + 2 * 1 + 4 * 0 = 2$
- $x_2 = t_{20} + 2t_{21} + 4t_{22} + 8t_{23}$ , donc  $x_2 = 1 * 0 + 2 * 1 + 4 * 0 + 8 * 0 = 2$
- $x_3 = t_{30} + 2t_{31}$ , donc  $x_3 = 1 * 0 + 2 * 0 = 0$
- $x_4 = t_{40} + 2t_{41} + 4t_{42} + 8t_{43} + 16t_{44}$ , donc  $x_4 = 1 * 0 + 2 * 0 + 4 * 1 + 8 * 1 + 16 * 0 = 12$
- $x_5 = t_{50} + 2t_{51} + 4t_{52}$ , donc  $x_5 = 1 * 0 + 2 * 0 + 4 * 0 = 0$

La solution reconstruite en entier est donc :(2, 2, 0, 12, 0)

### Comparaison de la valeur de l'optimum en 0-1 et en entier :

*Optimum en 0-1*

La fonction objectif est :

$$\begin{aligned} f(t) = & -32t_{10}t_{20} - 64t_{10}t_{21} - 128t_{10}t_{22} - 256t_{10}t_{23} - 4t_{10}t_{30} - 8t_{10}t_{31} - 32t_{10}t_{40} - \\ & 64t_{10}t_{41} - 128t_{10}t_{42} - 256t_{10}t_{43} - 512t_{10}t_{44} + 10t_{10}t_{50} + 20t_{10}t_{51} + 40t_{10}t_{52} - \\ & 64t_{11}t_{20} - 128t_{11}t_{21} - 256t_{11}t_{22} - 512t_{11}t_{23} - 8t_{11}t_{30} - 16t_{11}t_{31} - 64t_{11}t_{40} - \\ & 128t_{11}t_{41} - 256t_{11}t_{42} - 512t_{11}t_{43} - 1024t_{11}t_{44} + 20t_{11}t_{50} + 40t_{11}t_{51} + 80t_{11}t_{52} - \\ & 128t_{12}t_{20} - 256t_{12}t_{21} - 512t_{12}t_{22} - 1024t_{12}t_{23} - 16t_{12}t_{30} - 32t_{12}t_{31} - 128t_{12}t_{40} - \\ & 256t_{12}t_{41} - 512t_{12}t_{42} - 1024t_{12}t_{43} - 2048t_{12}t_{44} + 40t_{12}t_{50} + 80t_{12}t_{51} + 160t_{12}t_{52} - \\ & 26t_{20}t_{30} - 52t_{20}t_{31} - 16t_{20}t_{40} - 32t_{20}t_{41} - 64t_{20}t_{42} - 128t_{20}t_{43} - 256t_{20}t_{44} - \\ & 28t_{20}t_{50} - 56t_{20}t_{51} - 104t_{20}t_{52} - 52t_{21}t_{30} - 104t_{21}t_{31} - 32t_{21}t_{40} - 64t_{21}t_{41} - \\ & 128t_{21}t_{42} - 256t_{21}t_{43} - 512t_{21}t_{44} - 56t_{21}t_{50} - 112t_{21}t_{51} - 124t_{21}t_{52} - 104t_{22}t_{30} - \\ & 208t_{22}t_{31} - 64t_{22}t_{40} - 128t_{22}t_{41} - 256t_{22}t_{42} - 512t_{22}t_{43} - 1024t_{22}t_{44} - 112t_{22}t_{50} - \\ & 224t_{22}t_{51} - 448t_{22}t_{52} - 208t_{23}t_{30} - 416t_{23}t_{31} - 128t_{23}t_{40} - 256t_{23}t_{41} - 512t_{23}t_{42} - \\ & 1024t_{23}t_{43} - 2048t_{23}t_{44} - 224t_{23}t_{50} - 448t_{23}t_{51} - 996t_{23}t_{52} + 8t_{30}t_{40} + 16t_{30}t_{41} + \\ & 32t_{30}t_{42} + 64t_{30}t_{43} + 128t_{30}t_{44} + 6t_{30}t_{50} + 12t_{30}t_{51} + 24t_{30}t_{52} + 16t_{31}t_{40} + 32t_{31}t_{41} + \\ & 64t_{31}t_{42} + 128t_{31}t_{43} + 256t_{31}t_{44} + 12t_{31}t_{50} + 24t_{31}t_{51} + 48t_{31}t_{52} - 2t_{40}t_{50} - 4t_{40}t_{51} - \\ & 8t_{40}t_{52} - 4t_{41}t_{50} - 8t_{41}t_{51} - 16t_{41}t_{52} - 8t_{42}t_{50} - 16t_{42}t_{51} - 32t_{42}t_{52} - 16t_{43}t_{50} - \\ & 32t_{43}t_{51} - 64t_{43}t_{52} - 32t_{44}t_{50} - 64t_{44}t_{51} - 128t_{44}t_{52} + 13t_{10} + 26t_{11} + 52t_{12} - \\ & 34t_{20} - 68t_{21} - 136t_{22} - 272t_{23} + 3t_{30} + 6t_{31} - 24t_{40} - 48t_{41} - 96t_{42} - 192t_{43} - \\ & 384t_{44} - 30t_{50} - 60t_{51} - 120t_{52} \end{aligned}$$

Si  $t = (0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0)$

$$f(t) = -128 - 256 - 512 - 128 - 256 + 26 - 68 - 96 - 192$$

$$f(t) = -1610$$

*Optimum entier*

La fonction objectif est :

$$\begin{aligned} f(x) = & -32x_1x_2 - 4x_1x_3 - 32x_1x_4 + 10x_1x_5 - 26x_2x_3 - 16x_2x_4 - 28x_2x_5 + \\ & 8x_3x_4 + 6x_3x_5 - 2x_4x_5 + 13x_1 - 34x_2 + 3x_3 - 24x_4 - 30x_5 \end{aligned}$$

Si  $x = (2, 2, 0, 12, 0)$

$$f(x) = -32 * 2 * 2 - 32 * 2 * 12 - 16 * 2 * 12 + 13 * 2 - 34 * 2 - 24 * 12$$

$$f(x) = -1610$$

### A.3 Résolution par la méthode QCR :

Notre problème n'ayant pas de contraintes d'égalités, la résolution du programme SDP ne nous donne que des valeurs pour le vecteur  $\lambda$ .

La valeur du vecteur  $\lambda$  est (362.34, 725.16, 1450.77, 214.58, 429.74, 859.41, 1718.10, 1256.09, 2513.25, 106.86, 213.85, 427.44, 854.84, 1709.67, 489.50, 978.60, 1957.29) En rendant la fonction économique convexe grâce à la méthode QCR, notre problème peut être résolu par le solveur XPRESS, et on obtient les résultats suivants :

*Etude de la solution continue*

La solution continue est (0.27, 0.27, 0.27, 0.20, 0.20, 0.20, 0.20, 0, 0, 0.31, 0.31, 0.31, 0.31, 0.31, 0.01, 0.01, 0.01) et la valeur de la fonction économique et donc la borne est  $-5072.67$ , celle ci a été trouvée en 0 *secondes*

Les statistiques de la résolution continue sont les suivantes :

<i>Its</i>	<i>Primalobj.</i>	<i>Dualobj.</i>
0	$3.8343496e + 05$	$- 5.1331536e + 05$
1	$1.6041912e + 05$	$- 2.1809446e + 05$
2	$- 5.9029726e + 03$	$- 5.5594390e + 03$
3	$- 4.6038464e + 03$	$- 7.1772028e + 03$
4	$- 5.0392984e + 03$	$- 5.4011827e + 03$
5	$- 5.0722215e + 03$	$- 5.0879271e + 03$
6	$- 5.0726292e + 03$	$- 5.0728433e + 03$
7	$- 5.0726676e + 03$	$- 5.0726778e + 03$
8	$- 5.0726717e + 03$	$- 5.0726732e + 03$
9	$- 5.0726723e + 03$	$- 5.0726725e + 03$

*Etude de la solution en 0-1*

La solution en 0-1 est (0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0) et la valeur de la fonction économique est  $-1610$ , celle ci a été trouvée en 0 *secondes*.

Les statistiques de la résolution entière sont les suivantes :

<i>Node</i>	<i>BestSoln</i>	<i>BestBound</i>	<i>Gap</i>
20	$- 1295.000000$	$- 5072.672221$	74.47%
62	$- 1437.000000$	$- 3983.285400$	63.92%
65	$- 1492.000000$	$- 3983.285400$	62.54%
107	$- 1521.000000$	$- 2488.061768$	38.87%
124	$- 1610.000000$	$- 2174.976074$	25.98%

Le nombre de noeuds est 177 et XPRESS a trouvé 5 solutions en 0-1.

La solution reconstruite en entier est :(2, 2, 0, 12, 0)

## A.4 Résolution par la méthode "semi 0-1" :

Les valeurs du vecteur propre de la matrice  $Q$  sont les suivantes :

$spectre = (-29.49, -9.49, -4.66, 16.39, 27.24)$

La valeur minimum de ce spectre est :  $-29.49$ . Après la convexification, notre problème peut être résolu par le solveur XPRESS, et on obtient les résultats suivants :

### *Etude de la solution continue*

La solution continue est  $(1.64, 3.00, 0, 10.87, 0)$  et la valeur de la fonction économique et donc de la borne est :  $-6356.99$ , celle ci a été trouvée en  $0$  secondes

Les statistiques de la résolution continue sont les suivantes :

<i>Its</i>	<i>Primalobj.</i>	<i>Dualobj.</i>
0	$-5.6677953e + 03$	$-2.1110145e + 03$
1	$-1.0468150e + 04$	$-2.5636362e + 03$
2	$-1.0869566e + 04$	$-6.0088418e + 03$
3	$-1.5390798e + 04$	$-9.2370429e + 03$
4	$-1.3230392e + 04$	$-8.9101535e + 03$
5	$-1.0232446e + 04$	$-1.2355872e + 04$
6	$-9.5375927e + 03$	$-9.8701865e + 03$
7	$-8.6003468e + 03$	$-8.4583388e + 03$
8	$-6.1889571e + 03$	$-8.4910043e + 03$
9	$-6.2877753e + 03$	$-6.4370227e + 03$
10	$-6.3568645e + 03$	$-6.3571682e + 03$
11	$-6.3569874e + 03$	$-6.3569877e + 03$

### *Etude de la solution entière*

La solution entière est  $(2, 2, 0, 12, 0)$  et la valeur de la fonction économique est  $-1610$ , celle ci a été trouvée en  $0$  secondes.

Les statistiques de la résolution entière sont les suivantes :

<i>Node</i>	<i>BestSoln</i>	<i>BestBound</i>	<i>Gap</i>
14	$-1436.999996$	$-6356.987517$	77.39%
120	$-1450.99999$	$-2912.859131$	50.19%
133	$-1491.99999$	$-2837.054443$	47.41%
137	$-1536.99999$	$-2837.054443$	45.82%
190	$-1609.99998$	$-1897.226562$	15.14%

Le nombre de noeuds est 232 et XPRESS a trouvé 5 solutions entières.

## A.5 Résolution par la linéarisation :

Après la linéarisation, notre problème peut être résolu par le solveur XPRESS, et on obtient les résultats suivants :

### *Etude de la solution continue*

La solution continue est  $(2.01, 4.30, 0, 6.60, 0)$  et la valeur de la fonction économique et donc de la borne est :  $-3982.83$ , celle ci a été trouvée en *0 secondes*

Les statistiques de la résolution continue sont les suivantes :

<i>Its</i>	<i>Obj.Value</i>
0	- 15468.02026
37	- 3982.827869

### *Etude de la solution entière*

La solution entière est  $(2, 2, 0, 12, 0)$  et la valeur de la fonction économique est  $-1610$ , celle ci a été trouvée en *0 secondes*.

Les statistiques de la résolution entière sont les suivantes :

<i>Its</i>	<i>BestSoln</i>	<i>BestBound</i>	<i>Gap</i>
1	- 1359.000000	- 2839.599008	52.14%
2	- 1359.000000	- 2839.599008	52.14%
3	- 1359.000000	- 2267.826163	40.07%
4	- 1359.000000	- 2133.409643	36.30%

	<i>Node</i>	<i>BestSoln</i>	<i>BestBound</i>	<i>Gap</i>
8	- 1451.000000	- 2133.409643	31.99%	
10	- 1468.000000	- 2133.409668	31.19%	
16	- 1492.000000	- 2087.265869	28.52%	
23	- 1521.000000	- 1997.713135	23.86%	
28	- 1537.000000	- 1766.386475	12.99%	
32	- 1587.000000	- 1759.566040	9.81%	
34	- 1610.000000	- 1676.011719	3.94%	

Le nombre de noeuds est 35 et XPRESS a trouvé 9 solutions entières.

## B Détails des résultats

### B.1 Résolution des instances IQKP(1)

Légende :

- (1) : Résolution par la méthode de la plus petite valeur propre
- (2) : Résolution par la méthode QCR
- (3) : Résolution par la convexification "semi 0-1"
- (4) : Résolution par la linéarisation

On noteras les instances de la façon suivante :

IQKP(type d'instance)-numéro d'instance-n-[meilleure solution connue]

	nb_var	borne	gap	noeuds	tps	
IQKP(1)-1-5-[-1055]	(1)	15	-4357	312	103	0
	(2)	15	-2925	177	5	0
	(3)	25	-4441	320	173	0
	(4)	81	-2166	105	11	0
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-2-5-[-1610]	(1)	17	-7759	338	492	0
	(2)	17	-5072	215	124	0
	(3)	27	-6356	295	232	0
	(4)	87	-3982	147	35	0
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-3-5-[-3096]	(1)	16	-16821	443	327	0
	(2)	16	-10066	225	79	0
	(3)	26	-26498	755	381	0
	(4)	74	-7763	150	23	0
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-4-5-[-455]	(1)	14	-1365	200	15	0
	(2)	14	-880	93	51	0
	(3)	24	-1615	255	76	0
	(4)	75	-689	51	9	0
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-5-5-[-3497]	(1)	17	-13573	288	34	0
	(2)	17	-7627	118	19	0
	(4)	27	-56102	1504	1219	0
	(4)	77	-5197	48	1	0



	nb_var	borne	gap	noeuds	tps	
IQKP(1)-10-6-[-9214]	(1)	42	-131102	1313	1625860	1419
	(2)	42	-41273	348	12859	5
	(3)	62	-53110	476	50979	76
	(4)	337	-30329	229	63	0
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-10-7-[-7257]	(1)	38	-69446	857	474823	213
	(2)	38	-29008	299	7554	2
	(3)	58	-37524	417	5356	8
	(4)	317	-24987	244	97	0
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-10-8-[-6608]	(1)	41	-65465	890	1396950	1128
	(2)	41	-22815	245	3787	1
	(3)	61	-74322	1024	205052	254
	(4)	349	-17145	160	35	0
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-10-9-[-5443]	(1)	40	-70263	1190	1647008	1700
	(2)	40	-17867	228	1912	1
	(3)	60	-55654	922	243531	289
	(4)	348	-13724	148	53	0
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-10-10-[-1746]	(1)	37	-21184	1113	235661	71
	(2)	37	-8858	407	24665	8
	(3)	57	-9345	458	9260	13
	(4)	306	-7422	325	847	2
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-15-11-[-108438]	(1)	70	-919248	747	2230000	3600
	(2)	70	-265842	145	5839	7
	(3)	100	-490622	352	1470000	3600
	(4)	777	236001	117	23	0
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-15-12-[-53213]	(1)	72	-988167	1757	2860000	3600
	(2)	72	-228863	330	582730	680
	(3)	102	-470439	784	1290000	3600
	(4)	798	-185786	249	1576	3

	nb_var	borne	gap	noeuds	tps	
IQKP(1)-15-13-[-35033]	(1)	70	-797764	2177	3010000	3600
	(2)	70	-174664	399	645948	777
	(3)	100	-494490	1311	1960000	3600
	(4)	742	-102970	194	374	1
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-15-14-[-14062]	(1)	69	-296747	2010	3110000	3600
	(2)	69	-71066	405	1299453	2120
	(3)	99	-104311	641	1510000	3600
	(4)	783	-54836	290	493	3
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-15-15-[-19049]	(1)	70	-271023	1322	2960000	3600
	(2)	70	-68287	258	107284	97
	(3)	100	-121417	537	1450000	3600
	(4)	737	-53307	180	112	2
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-20-16-[-21137]	(1)	102	-1066334	4945	2170000	3600
	(2)	102	-161580	664	2390000	3600
	(3)	142	-619514	2830	850000	3600
	(4)	1372	-110874	424	8173	62
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-20-17-[-37737]	(1)	101	-1302540	3352	1670000	3600
	(2)	101	-216109	473	2210000	3600
	(3)	141	-776801	1958	1250000	3600
	(4)	1437	-136375	261	625	10
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-20-18-[-125453]	(1)	104	-2317536	1747	2110000	3600
	(2)	104	-411906	228	584320	1243
	(3)	144	-2540492	1925	1050000	3600
	(4)	1361	-283463	126	99	1
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-20-19-[-69057]	(1)	98	-1091830	1481	1910000	3600
	(2)	98	-224340	225	615332	1121
	(3)	138	-452569	555	1120000	3600
	(4)	1376	-180528	161	120	1

	nb_var	borne	gap	noeuds	tps	
IQKP(1)-20-20-[-42616]	(1)	100	-1483386	3381	1700000	3600
	(2)	100	-288516	577	2250000	3600
	(3)	140	-947733	2124	1360000	3600
	(4)	1357	-191404	349	9819	60
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-25-21-[-703749]	(1)	136	-14171844	1914	930000	3600
	(2)	136	-2227188	216	1190000	3600
	(3)	186	-4717436	570	600000	3600
	(4)	1452	-1747209	148	49	3
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-25-22-[-274530]	(1)	138	-11143119	3959	1050000	3600
	(2)	138	-1431397	421	1370000	3600
	(3)	188	-9767929	3458	480000	3600
	(4)	2251	-706125	157	547	21
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-25-23-[-261954]	(1)	141	-12327189	4606	860000	3600
	(2)	141	-1455643	456	1420000	3600
	(3)	191	-4514583	1623	680000	3600
	(4)	2350	-975886	273	552	20
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-25-24-[-105333]	(1)	137	-3289813	3023	1070000	3600
	(2)	137	-545832	418	1610000	3600
	(3)	187	-1638680	1456	800000	3600
	(4)	2258	-425390	303	1766	26
	nb_var	borne	gap	noeuds	tps	
IQKP(1)-25-25-[-270427]	(1)	140	-13944444	5056	990000	3600
	(2)	140	-1902998	604	1310000	3600
	(3)	190	-8971308	3217	420000	3600
	(4)	2233	-1039186	284	311	6

## B.2 Résolution des instances IQKP(2)

Légende :

- (1) : Résolution par la méthode de la plus petite valeur propre
- (2) : Résolution par la méthode QCR
- (3) : Résolution par la convexification "semi 0-1"
- (4) : Résolution par la linéarisation

On noteras les instances de la façon suivante :

IQKP(type d'instance)-numéro d'instance-n-[meilleure solution connue]

	nb_var	borne	gap	noeuds	tps	
IQKP(2)-1-5-[-658527]	(1)	30	-2165621	229	711595	1010
	(2)	30	-960471	46	77	0
	(3)	40	-833182	26	1216	3
	(4)	140	-767452	16	66	0
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-2-5-[-789628]	(1)	30	-2242535	184	617611	836
	(2)	30	-1085709	37	180	0
	(3)	40	-940748	19	947	2
	(4)	140	-953632	21	147	0
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-3-5-[-316870]	(1)	30	-1580573	430	1570000	3600
	(2)	30	-792207	150	1428	0
	(3)	40	-598170	88	50112	75
	(4)	140	-662910	109	103	0
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-4-5-[-784545]	(1)	30	-2017493	157	613312	901
	(2)	30	-1023701	30	204	0
	(3)	40	-853178	9	76	0
	(4)	140	-870337	11	32	0
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-5-5-[-203800]	(1)	30	-1231540	505	1480000	3600
	(2)	30	-365625	79	335	0
	(4)	40	-320330	57	570	1
	(4)	140	-468895	130	20	0

	nb_var	borne	gap	noeuds	tps	
IQKP(2)-10-6-[-2169930]	(1)	60	-7401656	241	1610000	3600
	(2)	60	-3543470	63	107417	216
	(3)	80	-2859631	32	29803	135
	(4)	455	-3272484	51	489	4
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-10-7-[-2593637]	(1)	60	-7425171	179	1720000	3600
	(2)	60	-3903163	50	13136	30
	(3)	80	-3155003	22	27687	160
	(4)	455	-3239797	25	113	1
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-10-8-[-1602132]	(1)	60	-5646348	252	1850000	3600
	(2)	60	-2439497	52	10671	31
	(3)	80	-2043044	27	12225	57
	(4)	455	-3257166	103	1423	8
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-10-9-[-2093842]	(1)	60	-7074564	237	1470000	3600
	(2)	60	-3427308	63	20197	46
	(3)	80	-2780884	33	4729	21
	(4)	455	-2898499	38	468	3
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-10-10-[-2221171]	(1)	60	-4511218	103	1440000	3600
	(2)	60	-3355805	51	19729	36
	(3)	80	-2654404	19	770000	3600
	(4)	455	-3201318	44	391	3
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-15-11-[-3752386]	(1)	90	-12994427	246	830000	3600
	(2)	90	-5578170	49	505888	2673
	(3)	120	-4693130	25	276690	2066
	(4)	945	-6538555	74	861	29
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-15-12-[-3343901]	(1)	90	-11800737	253	800000	3600
	(2)	90	-5361001	60	90757	530
	(3)	120	-4111533	23	39603	292
	(4)	945	-5835678	75	1741	49

	nb_var	borne	gap	noeuds	tps	
IQKP(2)-15-13-[-4004418]	(1)	90	-12080629	202	890000	3600
	(2)	90	-5674506	42	1150000	3600
	(3)	120	-4789365	20	480000	3600
	(4)	945	-7192847	80	3704	74
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-15-14-[-4261849]	(1)	90	-13797603	224	830000	3600
	(2)	90	-6086185	43	83277	567
	(3)	120	-5223472	23	100968	743
	(4)	945	-7336670	72	3836	190
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-15-15-[-3892800]	(1)	90	-10512803	170	850000	3600
	(2)	90	-6090054	56	148564	923
	(3)	120	-4033997	4	73	0
	(4)	945	-6482765	66	1958	127
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-20-16-[-4667621]	(1)	120	-19430012	316	470000	3600
	(2)	102	-8258031	77	340000	3600
	(3)	160	-6565588	41	340000	3600
	(4)	1610	-11530006	147	20000	3600
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-20-17-[-5315420]	(1)	120	-20717565	290	450000	3600
	(2)	120	-8924069	68	320000	3600
	(3)	160	-7131713	34	280000	3600
	(4)	1610	-11048246	107	20000	3600
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-20-18-[-5196027]	(1)	120	-18219285	250	510000	3600
	(2)	120	-10149835	96	320000	3600
	(3)	160	-6841580	32	300000	3600
	(4)	1610	-12178862	135	30000	3600
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-20-19-[-5901427]	(1)	120	-19337979	228	500000	3600
	(2)	120	-8672596	47	290000	3600
	(3)	160	-7188785	22	340000	3600
	(4)	1610	-11333093	92	44339	3600

	nb_var	borne	gap	noeuds	tps	
IQKP(2)-20-20-[-4961407]	(1)	120	-172906316	248	510000	3600
	(2)	120	-84565018	70	360000	3600
	(3)	160	-6285502	27	370000	3600
	(4)	1610	-10723331	116	20000	3600
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-25-21-[-8052753]	(1)	150	-28710143	256	310000	3600
	(2)	150	-12528838	55	210000	3600
	(3)	200	-10621661	32	250000	3600
	(4)	2450	-17479470	117	3862	3600
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-25-22-[-7897239]	(1)	150	-28323136	258	300000	3600
	(2)	150	-13036062	65	360000	3600
	(3)	200	-10097270	28	340000	3600
	(4)	2450	-17676054	124	10000	3600
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-25-23-[-8349313]	(1)	150	-26726655	220	270000	3600
	(2)	150	-12515185	50	190000	3600
	(3)	200	-10229068	22	270000	3600
	(4)	2450	-17936724	115	10124	3600
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-25-24-[-9407871]	(1)	150	-29877616	218	320000	3600
	(2)	150	-13430510	43	240000	3600
	(3)	200	-11516999	22	240000	3600
	(4)	2450	-18580270	97	10000	3600
	nb_var	borne	gap	noeuds	tps	
IQKP(2)-25-25-[-12611934]	(1)	150	-30798484	144	320000	3600
	(2)	150	-16995998	35	160000	3600
	(3)	200	-13574420	7	270000	3600
	(4)	2450	-19684442	56	25246	3600

### B.3 Résolution des instances UIQP(1)

Légende :

- (1) : Résolution par la méthode de la plus petite valeur propre
- (2) : Résolution par la méthode QCR
- (3) : Résolution par la convexification "semi 0-1"
- (4) : Résolution par la linéarisation

On noteras les instances de la façon suivante :

UIQP(type d'instance)-numéro d'instance-n-[meilleure solution connue]

	nb_var	borne	gap	noeuds	tps	
UIQP(1)-5-1-[-1813050]	(1)	30	-3318680	83	106576	109
	(2)	30	-2594991	43	87	0
	(3)	40	-1851323	2	43	0
	(4)	140	-1885097	4	1	0
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-5-2-[-1594600]	(1)	30	-3285921	106	393560	442
	(2)	30	-2314931	45	164	0
	(3)	40	-1696355	6	48	0
	(4)	140	-1634711	2	14	0
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-5-3-[-1930100]	(1)	30	-3516096	82	140000	143
	(2)	30	-2609913	35	550	0
	(3)	40	-2053519	6	83	0
	(4)	140	-1960037	2	7	0
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-5-4-[-777400]	(1)	30	-2030386	161	480000	645
	(2)	30	-1047509	35	768	0
	(3)	40	-800624	3	14	0
	(4)	140	-975523	25	15	0
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-5-5-[-549800]	(1)	30	-1234430	124	270000	280
	(2)	30	-704261	29	204	0
	(4)	40	-587581	7	781	2
	(4)	140	-820840	49	35	0



	nb_var	borne	gap	noeuds	tps	
UIQP(1)-10-6-[-2702072]	(1)	60	-5915007	119	1400000	3600
	(2)	60	-3816537	41	2463	7
	(3)	80	-2879139	7	6923	42
	(4)	455	-3348292	24	110	3
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-10-7-[-4160300]	(1)	60	-7771490	86	1440000	3600
	(2)	60	-5744410	38	959	3
	(3)	80	-4242638	2	25	0
	(4)	455	-4674481	12	37	1
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-10-8-[-1410656]	(1)	60	-4878325	246	1610000	3600
	(2)	60	-2373679	68	1639204	3392
	(3)	61	-1768782	25	53996	299
	(4)	455	-3601691	155	3436	32
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-10-9-[-2937050]	(1)	60	-7840617	167	1660000	3600
	(2)	60	-4587210	56	104974	202
	(3)	80	-3349829	14	4059	18
	(4)	455	-4262057	45	278	5
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-10-10-[-1490800]	(1)	60	-5831761	291	1700000	3600
	(2)	60	-2345477	57	97633	198
	(3)	57	-1811276	21	13223	55
	(4)	455	-3226005	116	235	10
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-15-11-[-3031400]	(1)	90	-13117620	332	1040000	3600
	(2)	90	-5480001	81	750000	3600
	(3)	120	-4186347	38	430000	3600
	(4)	945	-7560161	149	17477	814
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-15-12-[-5096150]	(1)	90	-15035903	195	1050000	3600
	(2)	90	-7816380	53	1030000	3600
	(3)	120	-5904418	16	430000	3600
	(4)	945	-8645962	70	2333	147

	nb_var	borne	gap	noeuds	tps	
UIQP(1)-15-13-[-3674600]	(1)	90	-11590018	215	910000	3600
	(2)	90	-5901774	61	910000	3600
	(3)	120	-4151195	13	10000	132
	(4)	945	-7414725	101	1216	95
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-15-14-[-4291077]	(1)	90	-11512115	168	890000	3600
	(2)	90	-6945878	62	870000	3600
	(3)	120	-4801604	12	440000	3600
	(4)	945	-8402376	95	6304	347
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-15-15-[-5321250]	(1)	90	-13397928	152	990000	3600
	(2)	90	-7903934	48	630000	3600
	(3)	120	-5725553	8	348655	3202
	(4)	945	-7968836	50	467	43
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-20-16-[-7797553]	(1)	120	-23101158	196	510000	3600
	(2)	102	-11721333	50	520000	3600
	(3)	160	-8712192	12	370000	3600
	(4)	1610	-13548927	74	3659	837
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-20-17-[-6426950]	(1)	120	-21018877	227	540000	3600
	(2)	120	-9999206	56	340000	3600
	(3)	160	-7440963	19	530000	3600
	(4)	1610	-12785116	98	11996	2658
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-20-18-[-6763200]	(1)	120	-19493435	188	550000	3600
	(2)	120	-10186845	50	450000	3600
	(3)	160	-7732319	14	430000	3600
	(4)	1610	-14220521	110	20000	3600
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-20-19-[-5788249]	(1)	120	-18166761	214	620000	3600
	(2)	120	-9197441	59	530000	3600
	(3)	160	-9468548	64	230000	3600
	(4)	1610	-12022198	107	22411	2933

	nb_var	borne	gap	noeuds	tps	
UIQP(1)-20-20-[-5591276]	(1)	120	-15041233	169	570000	3600
	(2)	120	-5890080	5	490000	3600
	(3)	160	-6878746	23	280000	3600
	(4)	1610	-11372736	103	20000	3600
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-25-21-[-10462391]	(1)	150	-31630971	202	330000	3600
	(2)	150	-16420986	57	490000	3600
	(3)	200	-12542939	20	350000	3600
	(4)	2450	-21604346	106	7938	3600
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-25-22-[-7478844]	(1)	150	-27035810	261	320000	3600
	(2)	150	-12429291	66	520000	3600
	(3)	200	-9358910	25	310000	3600
	(4)	2450	-20388942	172	10000	3600
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-25-23-[-6187310]	(1)	150	-25985944	320	340000	3600
	(2)	150	-10461688	69	450000	3600
	(3)	200	-8132615	31	350000	3600
	(4)	2450	-20173878	226	11751	3600
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-25-24-[-10426349]	(1)	150	-28869730	177	320000	3600
	(2)	150	-15913968	53	420000	3600
	(3)	200	-11391825	9	350000	3600
	(4)	2258	-20474330	97	6646	3600
	nb_var	borne	gap	noeuds	tps	
UIQP(1)-25-25-[-8175109]	(1)	150	-30700836	145	330000	3600
	(2)	150	-13920272	70	400000	3600
	(3)	200	-10520967	29	380000	3600
	(4)	2233	-20730307	65	10000	3600

## C Réalisation Pratique

### C.1 Détails de l'implémentation

Les résolutions de ces instances par les différentes méthodes ont été réalisées grâce au logiciel **Resolution** que nous avons codé en C. **Resolution** utilise également les logiciels Scilab [10], pour le calcul des valeurs propres, Sb [11], pour la résolution de programmes semi-définis, et XPress-Mosel [12] pour la résolution des programmes mathématiques convexifiés ou linéarisés. Le logiciel **Resolution** est composé des fichiers C suivants :

- **variables\_globales.h** et **variables\_globales.c** qui regroupent les entiers, vecteurs et matrices utilisés au cours du programme. Ces variables sont :
  - Les entiers :  $n$  (nombre de variable du problème),  $m$  (nombre de contraintes d'égalité),  $p$  (nombre de contraintes d'inégalité) et  $taille$  (nombre de variables du problème transformé en 0-1).
  - Les vecteurs :  $c$  (vecteur de la fonction objectif),  $C$  (vecteur de la fonction objectif transformé en 0-1),  $ui$  (vecteur des bornes des variables),  $lg$  (vecteur des logarithmes en base 2 des bornes des variables),  $lgc$  (vecteur cumulé des logarithmes en base 2 des bornes des variables),  $b$  (vecteur des contraintes d'égalité) et  $e$  (vecteur des contraintes d'inégalité).
  - Les matrices :  $q$  (matrice de la fonction objectif),  $Q$  (matrice de la fonction objectif transformée en 0-1),  $a$  (matrice des contraintes d'égalité),  $A$  (matrice des contraintes d'égalité transformée en 0-1),  $d$  (matrice des contraintes d'inégalité) et  $D$  (matrice des contraintes d'inégalité transformée en 0-1).
- **matrice.c** et **matrice.h** qui effectuent toutes les opérations nécessaires sur les matrices.
- **generation\_aleatoire.c** et **generation\_aleatoire.h** qui traitent toute la partie de génération d'entiers, de vecteurs ou de matrices et en particulier la génération d'un problème du sac à dos quadratique.
- **entree\_sortie.h** et **entree\_sortie.c**, qui s'occupent à la fois de lire une instance sauvegardée et d'écrire tous les fichiers de données nécessaires

entre les différents logiciels, qui sont :

- Pour le logiciel Scilab, un fichier de données de la matrice dont il faut calculer le spectre, nommé *lambda.sce*.
- Pour le logiciel XPress, les fichiers de données suivant :
  - *donnee\_c.dat* (où sont stockés les valeurs de  $n, m, p, ui, lg, lgc, c, C, b, e, q, Q, a, A, d, D$ ).
  - *lambda.dat* (contenant la valeur de la plus petite valeur propre de  $q$  ou de  $Q$ , calculée par Scilab).
  - *au.dat* (contenant la valeur du vecteur  $\lambda$  de convexification, calculé par Sb).
- Pour le logiciel Sb, le fichier *file\_c.sb* qui contient les valeurs de  $n, m, p, ui, lg, lgc, c, C, b, e, q, Q, a, A, d, D$ .
- Pour le logiciels **Resolution**, les fichiers de données suivant :
  - *spectre.txt* (généralisé par Scilab, il contient le spectre de la matrice  $q$  ou de  $Q$ ).
  - *au.txt* (généralisé par Sb, il contient la valeur du vecteur  $\lambda$  de convexification).
  - *result\_xpress.txt* (généralisé par XPress, il contient les statistiques de la résolution).
  - *sol.txt* (généralisé par XPress, il contient la valeur de la solution entière).
  - *sol\_0\_1.txt* (généralisé par XPress, il contient la valeur de la solution à valeurs dans  $\{0, 1\}$ ).
  - *sauvegarde\_inst.txt* (qui contient la sauvegarde de l'instance si elle vient d'être générée).
  - *sauvegarde.txt* (qui contient les statistiques et la solution de l'instance résolue).
- **passage\_0\_1.h** et **passage\_0\_1.c**, qui est chargé d'effectuer les opérations de transformations des coefficients des variables entières en ceux des variables à valeurs dans  $\{0, 1\}$  leurs correspondant.
- **reconstruction\_entier.h** et **reconstruction\_entier.c**, qui est chargé de transformer la solution à valeurs dans  $\{0, 1\}$ , en une solution à valeurs dans  $\mathbf{N}$ .
- **vp.h** et **vp.c**, qui résout l'instance en cours par la méthode de la plus petite valeur propre. Pour cela ils transforment d'abord le problème

en variables à valeurs dans  $\{0, 1\}$ , puis calculent la plus petite valeur propre de  $Q$ , avec le fichier **spectre.sce** qui est chargé dans Scilab, puis résolvent l'instance grâce au fichier **resolution\_vp.mos** qui est chargé dans XPress, et enfin reconstruisent la solution en valeurs entières.

- **qcr.h** et **qcr.c**, qui résout l'instance en cours par la méthode QCR. Pour cela ils transforment d'abord le problème en variables à valeurs dans  $\{0, 1\}$ , puis calculent le vecteur  $\lambda$  de convexification, avec le fichier **file\_c.sb** qui est chargé dans Sb, puis résolvent l'instance grâce au fichier **resolution\_qcr.mos** qui est chargé dans XPress, et enfin reconstruisent la solution en valeurs entières.
- **semi\_0\_1.h** et **semi\_0\_1.c**, qui résout l'instance en cours par la méthode "semi 0-1". Pour cela ils calculent la plus petite valeur propre de  $q$ , avec le fichier **spectre.sce** qui est chargé dans Scilab, puis résolvent l'instance grâce au fichier **resolution\_sem.mos** qui est chargé dans XPress.
- **linearisation.h** et **linearisation.c**, qui résout l'instance en cours par la linéarisation. Pour cela ils résolvent l'instance grâce au fichier **resolution\_lin.mos** qui est chargé dans XPress.
- **resolution.c**, qui est chargé de lancer le programme en fonction des options qui lui auront été fournies par l'utilisateur.

## C.2 Manuel de Resolution

### C.2.1 Compilation

Compiler l'ensemble des fichiers C avec la commande "make". Une commande "make clean" est disponible et permet d'effacer les fichiers créés pendant l'exécution de **Resolution**. Ne pas oublier de compiler les fichiers Mosel grâce aux lignes de commandes suivantes :

```
lambe_am@dept25:~/resolution> mosel
** Xpress-Mosel **
(c) Copyright Dash Associates 1998-2005
>cload resolution_vp.mos
Compiling 'resolution_vp.mos'...
>cload resolution_qcr.mos
Compiling 'resolution_qcr.mos'...
>cload resolution_sem.mos
Compiling 'resolution_sem.mos'...
>cload resolution_lin.mos
Compiling 'resolution_lin.mos'...
>q
Exiting.
```

### C.2.2 Utilisation

Les différentes options sont les suivantes :

- -g [sous-options], qui est le mode "génération aléatoire de l'instance".  
Details des sous-options (à séparer par des virgules) :
  - n : nombre de variables.
  - m : nombre de contraintes d'égalités.
  - p : nombre de contraintes d'inégalités.
  - min : valeur minimale des chiffres tirés aléatoirement.
  - med : valeur médium des chiffres tirés aléatoirement.
  - max : valeur maximale des chiffres tirés aléatoirement.
  - mult : multiplicateur des vecteurs contraintes.
  - nb\_inst : nombre d'instances à générer.
- -f [sous-options] qui est le mode "lecture de l'instance dans un fichier".  
Details des sous-options (à séparer par des virgules) :

- fichier : charger une instance contenue dans le fichier.
- num\_inst\_deb : charger à partir de l'instance numéro num\_inst\_dep.
- num\_inst\_fin : charger jusqu'à l'instance numéro num\_inst\_fin.

Les différents flags sont les suivants :

- -all, résolution par toutes les méthodes.
- -vp, résolution par la méthode de la plus petite valeur propre.
- -qcr, résolution par la méthode qcr.
- -sem, résolution par la méthode semi 0-1.
- -lin, résolution par la linéarisation.

Lors de la génération aléatoire, les coefficients de la matrice  $q$  et le vecteur  $c$  sont tirés entre les valeurs "min" et "med". Ceux des matrices  $a$  et  $d$  entre 1 et "max". Pour le vecteur  $b$ ,  $b_i = mult * \sum_{j=1}^n a_{ij}$  et pour le vecteur  $e$ ,

$$e_i = mult * \sum_{j=1}^n d_{ij}.$$

Les options ne peuvent être appelées simultanément, par contre, on peut utiliser plusieurs flags à la fois et dans l'ordre que l'on veut.

#### *Exemple d'une utilisation*

```
resolution -g n=5,m=2,p=1,min= -20,med=10,max=20,mult=10,nb_inst=4
--vp --qcr
resolution -f fichier="sauvegarde_inst.txt",num_inst_deb=4,num_inst_fin=8
--all
```

**Remarque :** Le générateur ne produisant, pour le moment, que des instances du sac à dos quadratique, il faut pour l'instant utiliser **Resolution** sans contraintes d'égalité, c'est à dire avec  $m = 0$ , car la probabilité d'obtenir un problème avec des contraintes d'égalités qui a une solution et qui est généré comme décrit précédemment est très faible. Une amélioration de **Resolution** qui générerait des problèmes avec des contraintes d'égalité est une extension prévue. De plus, la méthode QCR ne résout pour le moment que des problèmes du sac à dos quadratique.



### C.2.3 Format de lecture de fichier

Un fichier pouvant être lu par **Resolution** se présente de la manière suivante :

Instance 1

n=5,m=2,p=2

q=

( 0 -9 -2 -13 -4 )

( -9 0 -15 2 -4 )

( -2 -15 0 6 9 )

( -13 2 6 0 5 )

( -4 -4 9 5 0 )

c=

( 0 -38 -29 8 -23 )

ui=

( 20 20 26 38 40 )

a=

( 19 24 38 25 21 )

( 39 8 19 28 1 )

b=

( 127 95 )

d=

( 6 3 12 38 13 )

( 3 12 36 11 33 )

e=

( 72 95 )

**Remarque :** Les espaces et retours à la ligne sont essentiels à une bonne lecture. Il est prévu une amélioration de **Resolution**, qui lirait un format de fichier plus "standard".

## C.3 Détail du code

## variables\_globales.h/c

```
#ifndef VARIABLES_GLOBALES_H
#define VARIABLES_GLOBALES_H
int n;
int m;
int p;
int taille;
int ** q;
int **a;
int **d;
int *c;
int *ui;
int *b;
int * e;
int *lg;
int *lgc;
int ** Q;
int **A;
int **D;
int *C;
#endif
```

## matrice.h

```
#ifndef MATRICE_H
#define MATRICE_H
//gestion de la memoire
int* alloc_vecteur (int dimension);
int ** alloc_matrice (int ligne, int colonne);
double* alloc_vecteur_d (int dimension);
char* alloc_chaine (int dimension);

//operations basiques sur les vecteurs
int somme_vecteur(int * a, int n);
int maximum(int * vecteur,int n);
double minimum(double * vecteur,int n);
void copie_ligne(int ** B, int * v, int i, int t);
int nb_non_zero_matrice(int ** a, int n, int m);
int nb_non_zero_vecteur(int * v, int n);

//gestion des bornes en base 2
int log_base_2 (int x);
int taille_vecteur(int * ui, int n);
int * creation_ui_base2_cumule(int * ui, int n);
int * creation_ui_base2(int * ui, int n);
void insere_dans_vecteur(int deb,int fin, int * V,
                        int * C);
void insere_dans_matrice(int deb_l,int fin_l, int deb_c,
                        int fin_c, int ** A, int ** Q);
int * x_base2_V(int i, int borne_ui, int * C);
int ** x_base2_M(int i, int j, int borne_ui,
                int borne_uj,int ** Q);

//transformation d'un entier en caractere
char * intochar(int val);
#endif
```

## matrice.c

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<ctype.h>
#include"matrice.h"

//allocation memoire des matrice et des vecteurs
//d'entiers
int* alloc_vecteur (int dimension)
{
    int* vecteur = (int *)malloc(dimension*sizeof(int));
    return vecteur;
}

int ** alloc_matrice (int ligne, int colonne)
{
    int i;
    int** matrice = (int **)malloc(ligne*sizeof(int *));
    for (i=0;i<ligne;i++)
        matrice[i]= (int *)malloc(colonne*sizeof(int));
    return matrice;
}

//allocation memoire des vecteurs de doubles
double* alloc_vecteur_d (int dimension)
{
    double* vecteur = malloc(dimension*sizeof(double));
    return vecteur;
}

//allocation memoire des vecteurs de caracteres
char* alloc_chaine (int dimension)
{
```

```
    char* chaine = (char *)malloc((dimension+1)*
                                   sizeof(char));
    chaine[dimension]='\0';
    return chaine;
}

//transformation d'un entier en caractere
char * intochar(int val)
{
    int tmp1;
    int i=0;
    int tmp=val;
    int k=0;
    //i=log base 10 de val
    while(tmp>10)
    {
        tmp=(int)(tmp/10);
        i++;
    }
    char *s = alloc_chaine(i+1);
    tmp=val;
    while(i>=0)
    {
        tmp1=(int)(tmp/pow(10,i));
        s[k]=48+tmp1;
        tmp=tmp-tmp1*pow(10,i);
        i--;
        k++;
    }
    return s;
}

//operation basiques sur les vecteurs
int somme_vecteur(int * a, int n)
```

```

{
    int i;
    int b=0;
    for(i=0;i<n;i++)
        b+=a[i];
    return b;
}

int maximum(int * vecteur,int n)
{
    int i;
    int max;
    max = vecteur[0];
    for(i=1;i<n;i++)
        if(max < vecteur[i])
max = vecteur[i];
    return max;
}

double minimum(double * vecteur,int n)
{
    int i;
    double min;
    min = vecteur[0];
    for(i=1;i<n;i++)
        if(min > vecteur[i])
min = vecteur[i];
    return min;
}

void copie_ligne(int ** B, int * v, int i, int t)
{
    int j;
    for(j=0;j<t;j++)
        B[i][j] = v[j];
}

```

```

}

int nb_non_zero_vecteur(int * v, int n)
{
    int i;
    int c = 0;
    for(i=0;i<n;i++)
        if(v[i]!=0)
            c++;
    return c;
}

int nb_non_zero_matrice(int ** a, int n, int m)
{
    int j;
    int c = 0;
    for(j=0;j<m;j++)
        c+=nb_non_zero_vecteur(a[j],n);
    return c/2;
}

//gestion des bornes
//calcul du logarithme en base 2
int log_base_2 (int x)
{
    return (log(x)/log(2));
}

//calcul de la taille necessaire des coeficients en
// base 2
int taille_vecteur(int * ui, int n)
{
    int i,borne_ui;
    int taille = 0;
}

```

```

for(i=0;i<n;i++)
{
    borne_ui= (int)log_base_2(ui[i])+1;
    taille = taille + borne_ui;
}
return taille;
}

//vecteur des logs en base de des ui
int * creation_ui_base2(int * ui, int n)
{
    int * lgbis =alloc_vecteur(n);
    int i;
    for(i=0;i<n;i++)
        lgbis[i]= log_base_2(ui[i]) + 1;
    return lgbis;
}

//vecteur compose de la somme cumulee des logarithmes
// de ui en base 2
int * creation_ui_base2_cumule(int * ui, int n)
{
    int * lg = alloc_vecteur(n);
    int i;
    lg[0]= log_base_2(ui[0]) + 1;
    for(i=1;i<n;i++)
        lg[i] = lg[i-1] + log_base_2(ui[i]) + 1;
    return lg;
}

//insertion dans une matrice ou un vecteur
void insere_dans_vecteur(int deb,int fin, int * V,
    int * C)
{

```

```

    int i;
    int l=fin-deb;
    for(i=0;i<l;i++)
        C[i+deb]=V[i];
}

void insere_dans_matrice(int deb_l,int fin_l, int deb_c,
    int fin_c, int ** A, int ** Q)
{
    int i,j;
    int l=fin_l-deb_l;
    int c=fin_c-deb_c;
    for(i=0;i<l;i++)
        for(j=0;j<c;j++)
            Q[i+deb_l][j+deb_c]=A[i][j];
}

//transformation d'une case d'une matrice (ou d'un
//vecteur) en l'equivalent avec une variable en base 2
int * x_base2_V(int i, int borne_ui, int * C)
{
    int k;
    int * V = alloc_vecteur(borne_ui);
    for(k=0; k<borne_ui;k++)
        V[k]=(int)pow(2,k)*C[i];
    return V;
}

int ** x_base2_M(int i, int j, int borne_ui,
    int borne_uj, int ** Q)
{
    int k,l;
    int ** A = alloc_matrice(borne_ui,borne_uj);
    for(k=0; k<borne_ui;k++)

```

```

    for(l=0; l<borne_uj;l++)
A[k][l]=(int)pow(2,k+1)*Q[i][j];
return A;
}

```

#### generation\_aleatoire.h

```

#ifndef GENERATION_ALEATOIRE_H
#define GENERATION_ALEATOIRE_H

```

```

void generation_sac_a_dos(int min, int med,int max,
                          int mult);

```

```

#endif

```

#### generation\_aleatoire.c

```

#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<ctype.h>
#include<time.h>
#include"matrice.h"
#include"generation_aleatoire.h"
#include"variables_globales.h"

```

```

// genere un nombre aleatoire entre a et b (inclus)
int generation_entier(int a, int b)
{
    return (int) (a + rand()*((double)b-a+1)/
                 (RAND_MAX + 1.0));
}

```

```

//generation instance sac a dos quadratique
int * generation_vecteur(int a,int b)
{
    int i;
    int * V=alloc_vecteur(n);
    for(i=0;i<n;i++)
        V[i] = generation_entier(a,b);
    return V;
}

```

```

}

int ** generation_matrice_Q(int a, int b)
{
    int i,j;
    int ** M=alloc_matrice(n,n);
    for(i=0;i<n;i++)
        for(j=i;j<n;j++)
            if(i ==j)
M[i][j] = 0;
            else
{
M[i][j]=generation_entier(a,b)/2;
M[j][i]=M[i][j];
}
    return M;
}

int ** generation_matrice_contraintes(int nb_cont,
                                     int a, int b)
{
    int i,j;
    int ** M=alloc_matrice(nb_cont,n);
    for(i=0;i<nb_cont;i++)
        for(j=0;j<n;j++)
            M[i][j]=generation_entier(a,b);
    return M;
}

int * generation_vecteur_contraintes(int nb_cont,
                                     int ** M )
{
    int i;
    int * v=alloc_vecteur(m);
    for (i=0;i<nb_cont;i++)

```

```

        v[i]=somme_vecteur(M[i],n);
    return v;
}

int * generation_vecteur_ui(int mult, int nb_cont)
{
    int i;
    int * ui=alloc_vecteur(n);
    for(i=0;i<n;i++)
        ui[i]=mult * (int)(e[0]/d[0][i]);
    return ui;
}

void generation_sac_a_dos(int min, int med,int max,
                        int mult)
{
    srand(time(NULL));
    //generation des entiers, vecteurs et matrices donnees
    //du probleme
    q = generation_matrice_Q(min,med);
    c = generation_vecteur(min,med);
    if (m > 0)
    {
        a = generation_matrice_contraintes(m,1,max);
        b = generation_vecteur_contraintes(m,a);
    }
    else
    {
        a=alloc_matrice(1,1);
        a[0][0]=0;
        b=alloc_vecteur(1);
        b[0]=0;
    }
    if (p > 0)
    {

```

```

    d = generation_matrice_contraintes(p,1,max);
    e = generation_vecteur_contraintes(p,d);
    if (p == 1)
ui = generation_vecteur_ui(mult,p);
    else
ui = generation_vecteur(med,max);
    lgc=creation_ui_base2_cumule(ui,n);
    lg=creation_ui_base2(ui,n);
    taille = taille_vecteur(ui, n);
}
else
{
    d=alloc_matrice(1,1);
    d[0][0]=0;
    e=alloc_vecteur(1);
    e[0]=0;
    ui = generation_vecteur(med,max);
    lgc=creation_ui_base2_cumule(ui,n);
    lg=creation_ui_base2(ui,n);
    taille = taille_vecteur(ui, n);
}
}

```

## entree\_sortie.h

```

#ifndef ENTREE_SORTIE_H
#define ENTREE_SORTIE_H

//ecriture de vecteurs dans un fichier
void ecrire_vecteur(int * v, int n,FILE * temp);
void ecrire_vecteur_d(double * v, int n,FILE * temp);

//lecture de la solution
double * scan_vecteur_solution_N();
int * scan_vecteur_solution_0_1();

//ecriture des fichiers de sauvegarde et de donnees
void ecrire_fichier_instance(int num_inst );
void ecrire_fichier_instance_modifie( );
void ecrire_fichier_xpress( );
void ecrire_fichier_xpress_modifie( );
void ecrire_lambda_xpress();
void ecrire_au_xpress();
void ecrire_fichier_scilab(int ** M, int u);
void ecrire_fichier_sb(int t);
void ecrire_fichier_sauv(double * sol_N,int numero);
void ecrire_fichier_sauv_modifie(int * sol_0_1,
                                int * sol_N,int numero);

//copie d'un fichier dans un autre
void cat_sauvegarde(int numero);

//lecture du fichier d'instance
void lecture_fichier(char * file_name, char * num_inst);
#endif

```

## entree\_sortie.c

```

#include<stdio.h>

```



```

#include<math.h>
#include<stdlib.h>
#include<ctype.h>
#include<string.h>
#include"matrice.h"
#include"entree_sortie.h"
#include"variables_globales.h"

//nom des fichiers utilises entre les logiciels
static char donnee_xpress[13] = "donnee_c.dat";
static char donnee_lambda[11] = "lambda.dat";
static char donnee_au[7] = "au.dat";
static char donnee_scilab[11] = "lambda.sce";
static char donnee_sb[10] = "file_c.sb";
static char sol[8] = "sol.txt";
static char sol_0_1[12] = "sol_0_1.txt";
static char au[7] = "au.txt";
static char spectre[12] = "spectre.txt";
static char res[19]="result_xpress.txt";
static char sauv[15] = "sauvegarde.txt";
static char sauv_inst[21] = "sauvegarde_inst.txt";

//ecriture de vecteurs et matrices dans un fichier
void ecrire_vecteur(int * v, int n,FILE * temp)
{
    int i;
    fprintf(temp,"( ");
    for (i=0;i<n;i++)
        fprintf(temp,"%d ",v[i]);
    fprintf(temp," )\n");
    fprintf(temp,"\n");
}

void ecrire_vecteur_d(double * v, int n,FILE * temp)
{

```

```

    int i;
    fprintf(temp,"(");
    for (i=0;i<n;i++)
        fprintf(temp,"%d ",(int)v[i]);
    fprintf(temp," )\n");
    fprintf(temp,"\n");
}

void ecrire_matrice(int ** M, int m, int n,FILE *temp)
{
    int i,j;
    for (i=0;i<m;i++)
        {
            fprintf(temp,"( ");
            for(j=0;j<n;j++)
                fprintf(temp,"%d ", M[i][j]);
            fprintf(temp," )\n");
        }
    fprintf(temp,"\n");
}

//ecriture dans le fichier de sauvegarde des instances
void ecrire_fichier_instance(int num_inst )
{
    FILE * file_instance = fopen(sauv_inst,"a");
    fprintf(file_instance,"Instance %d\n",num_inst);
    fprintf(file_instance,"n=%d,m=%d,p=%d\n",n,m,p);
    fprintf(file_instance,"q=\n");
    ecrire_matrice(q,n,n,file_instance);
    fprintf(file_instance,"c=\n");
    ecrire_vecteur(c,n,file_instance);
    fprintf(file_instance,"ui=\n");
    ecrire_vecteur(ui,n,file_instance);
    if(m!=0)

```

```

{
    fprintf(file_instance,"a=\n");
    ecrire_matrice(a,m,n,file_instance);
    fprintf(file_instance,"b=\n");
    ecrire_vecteur(b,m,file_instance);
}
if(p!=0)
{
    fprintf(file_instance,"d=\n");
    ecrire_matrice(d,p,n,file_instance);
    fprintf(file_instance,"e=\n");
    ecrire_vecteur(e,p,file_instance);
}
fclose(file_instance);
}

void ecrire_fichier_instance_modifie()
{
    FILE * file_instance = fopen(sauv_inst,"a");
    fprintf(file_instance,"Q=\n");
    ecrire_matrice(Q,taille,taille,file_instance);
    if(m!=0)
    {
        fprintf(file_instance,"A=\n");
        ecrire_matrice(A,m,taille,file_instance);
    }
    if(p!=0)
    {
        fprintf(file_instance,"D=\n");
        ecrire_matrice(D,p,taille,file_instance);
    }
    fprintf(file_instance,"C=\n");
    ecrire_vecteur(C,taille,file_instance);
    fclose(file_instance);
}

```

```

//écriture dans les fichiers de données pour XPress
void ecrire_vecteur_xpress(int *v, int n,
                           FILE * temp, char * name)
{
    int i;
    fprintf(temp,"%s: [",name);
    for(i=0;i<n;i++)
        fprintf(temp,"%d ",v[i]);
    fprintf(temp,"]");
}

void ecrire_vecteur_xpress_d(double *v, int n,
                             FILE * temp, char * name)
{
    int i;
    fprintf(temp,"%s: [",name);
    for(i=0;i<n;i++)
        fprintf(temp,"%lf ",v[i]);
    fprintf(temp,"]");
}

void ecrire_matrice_xpress(int ** M, int m,int n,
                           FILE * temp, char * name)
{
    int i;
    int j;
    fprintf(temp,"%s: [",name);
    for(i=0;i<m;i++)
        for(j=0;j<n;j++)
            fprintf(temp,"%d ",M[i][j]);
    fprintf(temp,"]");
}

```

```

void ecriture_fichier_xpress( )
{
    int maxlg;
    FILE * file_xpress = fopen(donnee_xpress,"w");
    fprintf(file_xpress,"N:%d\n",n);
    ecriture_matrice_xpress(q,n,n,file_xpress,"Q");
    fprintf(file_xpress,"\n");
    ecriture_vecteur_xpress(c,n,file_xpress,"c");
    fprintf(file_xpress,"\n");
    ecriture_vecteur_xpress(lg,n,file_xpress,"lg");
    fprintf(file_xpress,"\n");
    ecriture_vecteur_xpress(ui,n,file_xpress,"ui");
    fprintf(file_xpress,"\n");
    if(m!=0)
    {
        fprintf(file_xpress,"m:%d\n",m);
        ecriture_matrice_xpress(a,m,n,file_xpress,"A");
        fprintf(file_xpress,"\n");
        ecriture_vecteur_xpress(b,m,file_xpress,"B");
        fprintf(file_xpress,"\n");
    }
    else
    {
        fprintf(file_xpress,"A:0\nB:0\n");
        fprintf(file_xpress,"m:1");
    }
    if(p!=0)
    {
        fprintf(file_xpress,"p:%d\n",p);
        ecriture_matrice_xpress(d,p,n,file_xpress,"D");
        fprintf(file_xpress,"\n");
        ecriture_vecteur_xpress(e,p,file_xpress,"E");
        fprintf(file_xpress,"\n");
    }
    else

```

```

    {
        fprintf(file_xpress,"D:0\nE:0\n");
        fprintf(file_xpress,"p:1");
    }
    maxlg=maximum(lg,n);
    fprintf(file_xpress,"maxlg:%d\n",maxlg);
    fclose(file_xpress);
}

void ecriture_fichier_xpress_modifie( )
{
    FILE * file_xpress = fopen(donnee_xpress,"w");
    fprintf(file_xpress,"N:%d\nn:%d\n",taille,n);
    ecriture_matrice_xpress(Q,taille,taille,file_xpress,"Q");
    fprintf(file_xpress,"\n");
    ecriture_vecteur_xpress(C,taille,file_xpress,"c");
    fprintf(file_xpress,"\n");
    ecriture_vecteur_xpress(lgc,n,file_xpress,"lgc");
    fprintf(file_xpress,"\n");
    ecriture_vecteur_xpress(ui,n,file_xpress,"ui");
    fprintf(file_xpress,"\n");
    if(m != 0)
    {
        fprintf(file_xpress,"m:%d\n",m);
        ecriture_matrice_xpress(A,m,taille,file_xpress,"A");
        fprintf(file_xpress,"\n");
        ecriture_vecteur_xpress(b,m,file_xpress,"B");
        fprintf(file_xpress,"\n");
    }
    else
    {
        fprintf(file_xpress,"A:0\nB:0\n");
        fprintf(file_xpress,"m:1");
    }
    if(p != 0)

```

```

{
    fprintf(file_xpress, "p:%d\n", p);
    ecriture_matrice_xpress(D, p, taille, file_xpress, "D");
    fprintf(file_xpress, "\n");
    ecriture_vecteur_xpress(e, p, file_xpress, "E");
    fprintf(file_xpress, "\n");
}
else
{
    fprintf(file_xpress, "D:0\nE:0\n");
    fprintf(file_xpress, "p:1");
}
fclose(file_xpress);
}

```

```
void ecriture_lambda_xpress()
```

```

{
    FILE * file_scilab, * file_xpress;
    double min;
    int a;
    //lecture de la plus petite valeur propre
    file_scilab = fopen(spectre, "r");
    fscanf(file_scilab, "%lf", &min);
    a = fclose(file_scilab);
    //ecriture de la plus petite valeur propre
    file_xpress = fopen(donnee_lambda, "w");
    fprintf(file_xpress, "lambda: %lf", min);
    a = fclose(file_xpress);
}

```

```
void ecriture_au_xpress()
```

```

{
    int i, n, tp;
    double * u;

```

```

FILE * file_au, * file_xpress;
file_au = fopen(au, "r");
file_xpress = fopen(donnee_au, "w");
fscanf(file_au, "%d", &n);
n = n - 2;
fscanf(file_au, "%d", &tp);
u = alloc_vecteur_d(n);
for(i=0; i<n; i++)
    fscanf(file_au, "%lf", &u[i]);
ecriture_vecteur_xpress_d(u, n, file_xpress, "u");
fclose(file_au);
fclose(file_xpress);
}

//ecriture dans le fichier de donnees pour Scilab
void ecriture_matrice_scilab(int ** M, int n, int m, FILE * temp)
{
    int i;
    int j;
    fprintf(temp, "[");
    for(i=0; i<n; i++)
    {
        for(j=0; j<m; j++)
        {
            fprintf(temp, "%d", M[i][j]);
            if(j!=m-1)
                fprintf(temp, ",");
        }
        if(i!=n-1)
            fprintf(temp, ";\n");
        else
            fprintf(temp, "]\n");
    }
}

```

```

void ecriture_fichier_scilab(int ** M, int u )
{
    FILE * file_scilab = fopen(donnee_scilab,"w");
    fprintf(file_scilab,"v=");
    ecriture_matrice_scilab(M,u,u,file_scilab);
    fprintf(file_scilab,";\n");
    fclose(file_scilab);
}

//ecriture du fichier de donnees pour Sb
void ecrire_matrice_Q_QSDP(FILE * temp,int t)
{
    int i,j;
    for(i=0;i<t;i++)
    {
        for(j=i;j<t;j++)
        if(Q[i][j]!=0)
            fprintf(temp,"%d %d %d\n",i+1,j+1,-Q[i][j]);
    }
}

void ecrire_vecteur_C_QSDP(FILE * temp,int t)
{
    int i;
    for(i=0;i<t;i++)
    {
        fprintf(temp,"0 ");
        fprintf(temp,"%d ", i+1);
        fprintf(temp,"%d\n", -C[i]/2);
    }
}

void ecriture_fichier_sb(int t)
{

```

```

FILE *fsdq;
int h,hbis,nb_cont,i,j,k;
h = nb_non_zero_matrice(Q,t,t);
hbis = h + t;
//ecriture du fichier d'entree de sb
fsdq=fopen(donnee_sb,"w");
fprintf(fsdq,"%d\n",t+1);
fprintf(fsdq,"SYMMETRIC_SPARSE\n");
fprintf(fsdq,"%d ",t+2);
fprintf(fsdq,"%d\n",hbis);
//ecriture de Q
ecrire_matrice_Q_QSDP(fsdq,t);
//ecriture de c
ecrire_vecteur_C_QSDP(fsdq,t);
//ecriture du nombre de contraintes
nb_cont=t+t*m+m+1+p;
fprintf(fsdq,"%d\n", nb_cont);
//ecriture des contraintes -bkxi + sum akjXij
for (i=0;i<m;i++)
    {
        for(j=0;j<t;j++)
        {
            fprintf(fsdq,"LOWRANK_SPARSE_DENSE\n");
            fprintf(fsdq,"%d", t+2 );
            fprintf(fsdq," 1 1\n");
            fprintf(fsdq,"%d", j+1);
            fprintf(fsdq," 0 1.\n");
            fprintf(fsdq,"%d", t+2);
            fprintf(fsdq," 1\n");
            fprintf(fsdq,"%d\n", -b[i]);
            for (k=0;k<t;k++)
                fprintf(fsdq,"%d\n",A[i][k]);

            fprintf(fsdq,"0\n");

```

```

fprintf(fsdqp,"= 0\n");
}
}
//écriture des contraintes Xii = xi
for (i=0;i<t;i++)
{
    fprintf(fsdqp,"SYMMETRIC_SPARSE\n");
    fprintf(fsdqp,"%d", t+2);
    fprintf(fsdqp," 2\n");
    fprintf(fsdqp,"0 ");
    fprintf(fsdqp,"%d", i+1);
    fprintf(fsdqp," -1\n");
    fprintf(fsdqp,"%d ", i+1);
    fprintf(fsdqp,"%d ", i+1);
    fprintf(fsdqp,"2.\n");
    fprintf(fsdqp,"= 0\n");

}
//écriture des contraintes d'egalite *2
if(m != 0)
    for (i=0;i<m;i++)
        {
fprintf(fsdqp,"SYMMETRIC_SPARSE\n");
fprintf(fsdqp,"%d ", t+2);
fprintf(fsdqp,"%d\n", t);
for (j=0;j<t;j++)
    {
        fprintf(fsdqp,"0 ");
        fprintf(fsdqp,"%d ",j+1 );
        fprintf(fsdqp,"%d\n",A[i][j]);
    }
fprintf(fsdqp,"= ");
fprintf(fsdqp,"%d\n",2*b[i]);
    }
//écriture des contraintes d'inegalite *2

```

```

if( p != 0)
    for (i=0;i<p;i++)
        {
fprintf(fsdqp,"SYMMETRIC_SPARSE\n");
fprintf(fsdqp,"%d ", t+2);
fprintf(fsdqp,"%d\n", t);
for(j=0;j<t;j++)
    {
        fprintf(fsdqp,"0 ");
        fprintf(fsdqp,"%d ",j+1 );
        fprintf(fsdqp,"%d\n",D[i][j]);
    }
fprintf(fsdqp,"< ");
fprintf(fsdqp,"%d\n",2*e[i]);
    }
//ECRITURE DE X11 = 1
fprintf(fsdqp,"SINGLETON\n");
fprintf(fsdqp,"%d ", t+2);
fprintf(fsdqp,"0 0 1\n");
fprintf(fsdqp,"= 1\n");
fclose(fsdqp);
}

//écriture du fichier de sauvegarde des resultats

void ecrire_fichier_sauv(double * sol_N,int numero)
{
    FILE * file_sauvegarde;
    file_sauvegarde = fopen(sauv,"a");
    fprintf(file_sauvegarde,"Instance %d\n",numero);
    fprintf(file_sauvegarde,"n :%d\nui :",n);
    ecrire_vecteur(ui,n,file_sauvegarde);
    fprintf(file_sauvegarde,"solution N :");
    ecrire_vecteur_d(sol_N,n,file_sauvegarde);
}

```

```

    fprintf(file_sauvegarde, "\n");
    fclose(file_sauvegarde);
}

void ecrire_fichier_sauv_modifie(int * sol_0_1,
                                int * sol_N, int numero)
{
    FILE * file_sauvegarde;
    file_sauvegarde = fopen(sauv, "a");
    fprintf(file_sauvegarde, "Instance %d\n", numero);
    fprintf(file_sauvegarde, "n :%d\n\nui : \n", n);
    ecrire_vecteur(ui, n, file_sauvegarde);
    fprintf(file_sauvegarde, "solution 0_1 :");
    ecrire_vecteur(sol_0_1, taille, file_sauvegarde);
    fprintf(file_sauvegarde, "solution N :");
    ecrire_vecteur(sol_N, n, file_sauvegarde);
    fprintf(file_sauvegarde, "\n");
    fclose(file_sauvegarde);
}

//copie d'un fichier dans un autre
void cat_sauvegarde(int numero)
{
    FILE * file_sauvegarde, * file_res;
    char c;
    file_sauvegarde = fopen(sauv, "a");
    file_res = fopen(res, "r");
    fprintf(file_sauvegarde, "stat XPress %d\n", numero);
    c = fgetc(file_res);
    while (c != -1)
    {
        fputc(c, file_sauvegarde);
        c = fgetc(file_res);
    }
    fclose(file_res);
    fclose(file_sauvegarde);
}

```

```

}

//lecture de la solution
double * scan_vecteur_solution_N()
{
    int j;
    FILE * file_sol;
    double * v = alloc_vecteur_d(n);
    file_sol = fopen(sol, "r");
    for(j=0; j<n; j++)
        fscanf(file_sol, "%lf", &v[j]);
    fclose(file_sol);
    return v;
}

int * scan_vecteur_solution_0_1()
{
    int j;
    FILE * file_sol_0_1;
    int * v = alloc_vecteur(taille);
    file_sol_0_1 = fopen(sol_0_1, "r");
    for(j=0; j<taille; j++)
        fscanf(file_sol_0_1, "%d", &v[j]);
    fclose(file_sol_0_1);
    return v;
}

//lecture des instances
int scan_entier_nomme(FILE * temp, char k)
{
    int g;
    char c = getc(temp);
    while (c != k)
        c = getc(temp);
}

```

```

    c= getc(temp);
    fscanf(temp,"%d",&g);
    return g;
}

int * scan_vecteur_nomme(FILE * temp, int t,char d)
{
    char c = getc(temp);
    int i;
    int * v= alloc_vecteur(t);
    while (c != d)
        c= getc(temp);
    while (c != '(')
        c= getc(temp);
    for(i=0;i<t;i++)
        fscanf(temp,"%d",&v[i]);
    return v;
}

int ** scan_matrice_nomme(FILE * temp,int t1, int t2,
                           char d)
{
    char c = getc(temp);
    int j;
    int ** M= alloc_matrice(t2,t1);
    while (c != d)
        c= getc(temp);
    while (c != '(')
        c= getc(temp);
    for(j=0;j<t1;j++)
        fscanf(temp,"%d",&M[0][j]);
    for(j=1;j<t2;j++)
        M[j]=scan_vecteur_nomme(temp,t1,')');
    return M;
}

```

```

void lecture_fichier(char * file_name, char * num_inst)
{
    FILE *file_inst;
    char * s = alloc_chaine(13);
    char *test;
    test = alloc_chaine(13);
    strcpy(test,"Instance ");
    strcat(test,num_inst);
    strcat(test,"\n");
    //lecture du fichier d'instances
    file_inst = fopen(file_name,"r");
    fgets(s,13,file_inst);
    while(strcmp(s,test) != 0)
        fgets(s,13,file_inst);
    n=scan_entier_nomme(file_inst,'n');
    m=scan_entier_nomme(file_inst,'m');
    p=scan_entier_nomme(file_inst,'p');
    q = scan_matrice_nomme(file_inst,n,n,'q');
    c = scan_vecteur_nomme(file_inst,n,'c');
    ui = scan_vecteur_nomme(file_inst,n,'u');
    lgc = creation_ui_base2_cumule(ui,n);
    lg = creation_ui_base2(ui,n);
    if(m>0)
    {
        a= scan_matrice_nomme(file_inst,n,m,'a');
        b= scan_vecteur_nomme(file_inst,m,'b');
    }
    else
    {
        a=alloc_matrice(1,1);
        a[0][0]=0;
        b=alloc_vecteur(1);
        b[0]=0;
    }
}

```



```

if(p>0)
{
    d= scan_matrice_nomme(file_inst,n,p,'d');
    e= scan_vecteur_nomme(file_inst,p,'e');
}
else
{
    d=alloc_matrice(1,1);
    d[0][0]=0;
    e=alloc_vecteur(1);
    e[0]=0;
}
taille = taille_vecteur(ui,n);
fclose(file_inst);
}

```

### passage\_0.1.h

```

#ifndef PASSAGE_0_1_H
#define PASSAGE_0_1_H

```

```

void passage_0_1();
#endif

```

### passage\_0.1.c

```

#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<ctype.h>
#include"passage_0_1.h"
#include"matrice.h"
#include"entree_sortie.h"
#include"generation_aleatoire.h"
#include"variables_globales.h"

```

```

//creation des vecteurs et matrices finaux

```

```

int * creation_V_base2(int * v)
{
    int i,fin,borne_ui;
    int borne_ui_prec = 0;
    int deb = 0;
    int * D;
    int * V = alloc_vecteur(taille);
    D=alloc_vecteur(lg[0]);
    D = x_base2_V(0,lg[0],v);
    insere_dans_vecteur(0,lg[0],D,V);
    for(i=1;i<n;i++)
    {
        D = alloc_vecteur(lg[i]);
        deb = lgc[i-1];
    }
}

```

```

        fin = lgc[i];
        D = x_base2_V(i,lg[i],v);
        insere_dans_vecteur(deb,fin,D,V);
    }
    return V;
}

int ** creation_Q_base2(int ** M)
{
    int i,j, fin_l, fin_c, borne_ui, borne_uj;
    int deb_l = 0;
    int deb_c = 0;
    int borne_ui_prec = 0;
    int borne_uj_prec = 0;
    int ** A;
    int ** B = alloc_matrice(taille,taille);
    for(i=0;i<n;i++)
    {
        borne_ui = (int)log_base_2(ui[i])+1;
        deb_l = deb_l + borne_ui_prec;
        fin_l = deb_l + borne_ui;
        for(j=0;j<n;j++)
        {
            borne_uj = (int)log_base_2(ui[j])+1;
            A = alloc_matrice(borne_ui,borne_uj);
            deb_c = deb_c + borne_uj_prec;
            fin_c = deb_c + borne_uj;
            A = x_base2_M(i,j,borne_ui,borne_uj,M);
            insere_dans_matrice(deb_l,fin_l,deb_c,fin_c,A,B);
            borne_uj_prec=borne_uj;
        }
    }
    borne_uj_prec=0;
    deb_c=0;
    borne_ui_prec=borne_ui;
}

```

```

    }
    return B;
}

int ** creation_A_base2(int ** M, int nb_cont)
{
    int i;
    int ** B = alloc_matrice(nb_cont,taille);
    int *v;
    v=creation_V_base2(M[0]);
    copie_ligne(B,v,0,taille);
    for(i=1;i<nb_cont;i++)
    {
        v=creation_V_base2(M[i]);
        copie_ligne(B,v,i,taille);
    }
    return B;
}

void passage_0_1( )
{
    //Modification des donnees du probleme entier en probleme en 0-1
    Q = creation_Q_base2(q);
    C = creation_V_base2(c);
    if(m!=0)
        A = creation_A_base2(a,m);
    else
    {
        A=alloc_matrice(1,1);
        A[0][0]=0;
    }
    if(p!=0)
        D = creation_A_base2(d,p);
    else
    {

```

```

    D=alloc_matrice(1,1);
    D[0][0]=0;
}
}

```

#### reconstruction\_entier.h

```

#ifndef RECONSTRUCTION_ENTIER_H
#define RECONSTRUCTION_ENTIER_H

void reconstruction_solution(int numero );
#endif

```

#### reconstruction\_entier.c

```

#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include"matrice.h"
#include"reconstruction_entier.h"
#include"entree_sortie.h"
#include"generation_aleatoire.h"
#include"variables_globales.h"

//reconstruction du vecteur solution en nombre entiers
//a partir de la solution xpress
int * reconstruction(int * sol_0_1)
{
    int * solution_N = alloc_vecteur(n);
    int i,j,log_ui;
    int dep_j=0;
    for(i=0;i<n;i++)
    {
        log_ui=(int)log_base_2(ui[i])+1;
        solution_N[i]=0;
        for(j=dep_j;j<log_ui+dep_j;j++)
        solution_N[i] = solution_N[i]+ (int)pow(2,j-dep_j)*
                                                                sol_0_1[j];
        dep_j=dep_j+log_ui;
    }
}

```

```

    return solution_N;
}

void reconstruction_solution(int numero)
{
    int * sol_0_1;
    int * sol_N;
    sol_0_1 = scan_vecteur_solution_0_1();
    sol_N=reconstruction(sol_0_1);
    ecrire_fichier_sauv_modifie(sol_0_1,sol_N,numero);
    cat_sauvegarde(numero);
}

```

## vp.h

```

#ifndef VP_H
#define VP_H

```

```

void vp(int numero);
#endif

```

## vp.c

```

#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include<math.h>
#include<stdlib.h>
#include<ctype.h>
#include <sys/wait.h>
#include"matrice.h"
#include"entree_sortie.h"
#include"reconstruction_entier.h"
#include"passage_0_1.h"
#include"variables_globales.h"

```

```

void vp(int numero)
{
    int file_sol;
    pid_t pid_fils;
    passage_0_1();
    ecrire_fichier_xpress_modifie( );
    ecrire_fichier_scilab(Q, taille);
    pid_fils=fork();
    if (pid_fils==0)
        //fils exec

```

```

    execlp("scilab","scilab","-nw","-f","spectre.sce"
          ,NULL);
else
    //pere wait pid
    wait(NULL);
ecriture_lambda_xpress();
pid_fils=fork();
if (pid_fils==0)
    //fils exec
    {
        file_sol = open("result_xpress.txt", O_RDWR |
            O_CREAT | O_TRUNC ,S_IRWXU | S_IRWXG | S_IRWXO);
        dup2(file_sol,STDOUT_FILENO);
        close(file_sol);
        execlp("mosel","mosel","-c",
            "exec_resolution_vp.mos",NULL);
    }
else
    //pere wait pid
    wait(NULL);
reconstruction_solution(numero);
}

```

## qcr.h

```

#ifndef QCR_H
#define QCR_H

```

```

void qcr(int numero);
#endif

```

## qcr.c

```

#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include<math.h>
#include<stdlib.h>
#include<ctype.h>
#include <sys/wait.h>
#include"matrice.h"
#include"entree_sortie.h"
#include"reconstruction_entier.h"
#include"passage_0_1.h"
#include"variables_globales.h"

```

```

void qcr(int numero)
{
    int file_sol;
    pid_t pid_fils;
    passage_0_1();
    ecriture_fichier_xpress_modifie( );
    ecriture_fichier_sb(taille);
    pid_fils=fork();
    if (pid_fils==0)
        //fils exec

```

```

    execlp("sb","sb","-oy","au.txt","-f","file_c.sb",
          NULL);
else
    //pere wait pid
    wait(NULL);
ecriture_au_xpress();
pid_fils=fork();
if (pid_fils==0)
    //fils exec
    {
        file_sol = open("result_xpress.txt", O_RDWR |
            O_CREAT | O_TRUNC ,S_IRWXU | S_IRWXG | S_IRWXO);
        dup2(file_sol,STDOUT_FILENO);
        close(file_sol);
        execlp("mosel","mosel","-c",
            "exec_resolution_qcr.mos",NULL);
    }
else
    //pere wait pid
    wait(NULL);
reconstruction_solution(numero);
}

```

## semi\_0\_1.h

```

#ifndef SEMI_0_1_H
#define SEMI_0_1_H

```

```

void semi_0_1(int numero);
#endif

```

## semi\_0\_1.c

```

#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include<math.h>
#include<stdlib.h>
#include<ctype.h>
#include <sys/wait.h>
#include"matrice.h"
#include"entree_sortie.h"
#include"reconstruction_entier.h"
#include"passage_0_1.h"
#include"variables_globales.h"

```

```

void semi_0_1(int numero)
{
    double * sol_N;
    int file_sol;
    pid_t pid_fils;
    ecriture_fichier_xpress( );
    ecriture_fichier_scilab(q,n);
    pid_fils=fork();
    if (pid_fils==0)
        //fils exec

```

```

    execlp("scilab","scilab","-nw","-f","spectre.sce"
          ,NULL);
else
    //pere wait pid
    wait(NULL);
    ecriture_lambda_xpress();
    pid_fils=fork();
    if (pid_fils==0)
        //fils exec
        {
            file_sol = open("result_xpress.txt", O_RDWR |
                O_CREAT | O_TRUNC ,S_IRWXU | S_IRWXG | S_IRWXO);
            dup2(file_sol,STDOUT_FILENO);
            close(file_sol);
            execlp("mosel","mosel","-c",
                "exec resolution_sem.mos",NULL);
        }
    else
        //pere wait pid
        wait(NULL);
        sol_N=scan_vecteur_solution_N();
        ecrire_fichier_sauv(sol_N,numero);
        cat_sauvegarde(numero);
}

```

## linearisation.h

```

#ifndef LINEARISATION_H
#define LINEARISATION_H

```

```

void linearisation(int numero);
#endif

```

## linearisation.c

```

#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include<math.h>
#include<stdlib.h>
#include<ctype.h>
#include <sys/wait.h>
#include"matrice.h"
#include"entree_sortie.h"
#include"reconstruction_entier.h"
#include"passage_0_1.h"
#include"variables_globales.h"

```

```

void linearisation(int numero)
{
    double * sol_N;
    int file_sol;
    pid_t pid_fils;
    ecriture_fichier_xpress( );
    pid_fils=fork();
    if (pid_fils==0)
        //fils exec
        {

```

```

    file_sol = open("result_xpress.txt", O_RDWR |
        O_CREAT | O_TRUNC ,S_IRWXU | S_IRWXG | S_IRWXO);
    dup2(file_sol,STDOUT_FILENO);
    close(file_sol);
    execlp("mosel","mosel","-c",
        "exec resolution_sem.mos",NULL);
}
else
    //pere wait pid
    wait(NULL);
sol_N=scan_vecteur_solution_N();
ecrire_fichier_sauv(sol_N,numero);
cat_sauvegarde(numero);
}

```

## resolution.c

```

#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<math.h>
#include<stdlib.h>
#include<ctype.h>
#include<sys/wait.h>
#include<getopt.h>
#include<assert.h>
#include<sys/resource.h>
#include<time.h>
#include<sys/time.h>
#include<sys/sysctl.h>
#include<string.h>
#include"matrice.h"
#include"entree_sortie.h"
#include"generation_aleatoire.h"
#include"reconstruction_entier.h"
#include"passage_0_1.h"
#include"variables_globales.h"
#include"vp.h"
#include"qcr.h"
#include"semi_0_1.h"
#include"linearisation.h"

static int flag_all;
static int flag_vp;
static int flag_qcr;
static int flag_sem;
static int flag_lin;
static char sauv[15]="sauvegarde.txt";

```



```

enum
{
    N,
    M,
    P,
    MIN,
    MED,
    MAX,
    MULT,
    NB_INST,
    FICHER,
    NUM_INST_DEB,
    NUM_INST_FIN,
    THE_END
};

char *sub_opts[] =
{
    [N] = "n",
    [M] = "m",
    [P] = "p",
    [MIN] = "min",
    [MED] = "med",
    [MAX] = "max",
    [MULT] = "mult",
    [NB_INST] = "nb_inst",
    [FICHER] = "fichier",
    [NUM_INST_DEB] = "num_inst_deb",
    [NUM_INST_FIN] = "num_inst_fin",
    [THE_END] = NULL
};

void usage(){
    printf("Usage: resolution -g [sous-options]

```

```

        (generation aleatoire de l'instance)\n");
printf("        -f [sous-options] (lecture de
        l'instance dans un fichier)\n");
printf("Attention on ne peut appeller qu'une des
        options g et f a la fois\n");
printf("        --all (resolution par toutes les
        methodes)\n ");
printf("        --vp (resolution par la methode
        de la plus petite valeur propre)\n ");
printf("        --qcr (resolution par la methode
        qcr)\n ");
printf("        --sem (resolution par la methode
        semi 0-1)\n ");
printf("        --lin (resolution par la
        linearisation)\n ");
printf(" Details des sous-options(a separees par des
        virgules):\n");
printf("        -g        n: nombre de variables\n");
printf("                m: nombre de contraintes d'egalites
                \n");
printf("                p: nombre de contraintes
                d'inegalites\n");
printf("                min: valeur min des chiffres tires
                aleatoirement\n");
printf("                med: valeur medium des chiffres
                tires aleatoirement\n");
printf("                max: valeur max des chiffres tires
                aleatoirement\n");
printf("                mult: multiplicateur des vecteurs
                contraintes\n");
printf("                nb_inst: nombre d'instances a
                generer\n");
printf("        -f        fichier: charger une instance
                contenue dans le fichier\n");
printf("                num_inst_deb: charger a partir de

```

```

        l'instance numero num_inst_dep\n");
printf("        num_inst_fin: charger jusqu'a
        l'instance numero num_inst_fin\n");
printf("exemples:\n");
printf("resolution -g n=5,m=2,p=1,min=-20,med=10,max=20
        ,mult=10,nb_inst=4 --vp\n");
printf("resolution -f fichier=\"sauvegarde_inst.txt\"
        ,num_inst_deb=2,num_inst_fin=5 --all\n\n");
}

int main (argc, argv)
    int argc;
    char **argv;
{
    FILE * fd,*file_sauv;
    char *subopts, *value;
    char *buf= alloc_chaine(20);
    int car,min,med,max,mult,nb_inst,i,num_inst_deb;
    int num_inst_fin;
    int g=0;
    int f=0;
    if(argc==1)
    {
        usage();
        exit(-1);
    }
    //lecture des options
    while(1)
    {
        static struct option long_options[] =
            { //flag --
{"all",  no_argument, &flag_all, 1},
{"vp",   no_argument, &flag_vp,  1},
{"qcr",  no_argument, &flag_qcr, 1},
{"sem",  no_argument, &flag_sem, 1},

```

```

{"lin",  no_argument, &flag_lin, 1},
//option -
{"generation",required_argument, 0, 'g'},
{"fichier",  required_argument, 0, 'f'},
{0, 0, 0, 0}
};
    // getopt_long stocke les index des options
    int option_index = 0;
    car = getopt_long (argc, argv, "g:f:",
        long_options, &option_index);
    // Detecte la fin des options
    if (car == -1)
break;
    switch (car)
    {
case 0:
    if (long_options[option_index].flag != 0)
        break;
    break;
    case 'g':
    g=1;
    subopts = optarg;
    while (*subopts != '\0')
        switch (getsubopt (&subopts, sub_opts, &value))
        {
    case N:
    if (value == NULL)
        abort ();
    n = atoi(value);
    break;
    case M:
    if (value == NULL)
        abort ();
    m = atoi(value);
    break;

```

```

        case P:
if (value == NULL)
    abort ();
p = atoi(value);
break;
        case MIN:
if (value == NULL)
    abort ();
min = atoi(value);
break;
        case MED:
if (value == NULL)
    abort ();
med = atoi(value);
break;
        case MAX:
if (value == NULL)
    abort ();
max = atoi(value);
break;
        case MULT:
if (value == NULL)
    abort ();
mult = atoi(value);
break;
        case NB_INST:
if (value == NULL)
    abort ();
nb_inst = atoi(value);
break;
    }
    break;
case 'f':
f=1;
subopts = optarg;

```

```

while (*subopts != '\0')
    switch (getsubopt (&subopts, sub_opts, &value))
    {
        case FICHER :
if (value == NULL)
    abort ();
strcpy(buf,value);
break;
        case NUM_INST_DEB:
if (value == NULL)
    abort ();
num_inst_deb = atoi(value);
break;
        case NUM_INST_FIN:
if (value == NULL)
    abort ();
num_inst_fin = atoi(value);
break;
    }
    break;
default:
    abort ();
}

}

if(g==1 && f==1)
{
    usage();
    return 1;
}

if(g)
    for(i=1;i<=nb_inst;i++)
    {
generation_sac_a_dos(min,med,max,mult);
ecrire_fichier_instance(i);
if(flag_all)

```

```

{
  file_sauv=fopen(sauv,"a");
  fprintf(file_sauv,"\n");
  fprintf(file_sauv,"Resolution par la valeur propre\n");
  fclose(file_sauv);
  vp(i);
  file_sauv=fopen(sauv,"a");
  fprintf(file_sauv,"\n");
  fprintf(file_sauv,"Resolution par qcr\n");
  fclose(file_sauv);
  qcr(i);
  file_sauv=fopen(sauv,"a");
  fprintf(file_sauv,"\n");
  fprintf(file_sauv,"Resolution par semi 0-1\n");
  fclose(file_sauv);
  semi_0_1(i);
  file_sauv=fopen(sauv,"a");
  fprintf(file_sauv,"\n");
  fprintf(file_sauv,"Resolution par la linearisation\n");
  fclose(file_sauv);
  linearisation(i);
  ecrire_fichier_instance_modifie();
}
  if(flag_vp)
  {
    file_sauv=fopen(sauv,"a");
    fprintf(file_sauv,"\n");
    fprintf(file_sauv,"Resolution par la valeur propre\n");
    fclose(file_sauv);
    vp(i);
    ecrire_fichier_instance_modifie();
  }
if(flag_qcr)
  {
    file_sauv=fopen(sauv,"a");

```

```

    fprintf(file_sauv,"Resolution par qcr\n");
    fprintf(file_sauv,"\n");
    fclose(file_sauv);
    qcr(i);
    if(!flag_vp)
      ecrire_fichier_instance_modifie();
  }
if(flag_semi)
  {
    file_sauv=fopen(sauv,"a");
    fprintf(file_sauv,"\n");
    fprintf(file_sauv,"Resolution par semi 0-1\n");
    fclose(file_sauv);
    semi_0_1(i);
  }
if(flag_lin)
  {
    file_sauv=fopen(sauv,"a");
    fprintf(file_sauv,"\n");
    fprintf(file_sauv,"Resolution par la linearisation\n");
    fclose(file_sauv);
    linearisation(i);
  }
  }
else
  {
    fd = fopen( buf, "r" );
    if (fd==NULL)
  {
    printf("erreur ouverture\n");
    usage();
    return 1;
  }

    fclose(fd);
    for(i=num_inst_deb;i<=num_inst_fin;i++)

```

```

{
lecture_fichier(buf,intochar(i));
if(flag_all)
{
printf("dedans\n");
file_sauv=fopen(sauv,"a");
fprintf(file_sauv,"\n");
fprintf(file_sauv,"Resolution par la valeur propre\n");
fclose(file_sauv);
vp(i);
file_sauv=fopen(sauv,"a");
fprintf(file_sauv,"\n");
fprintf(file_sauv,"Resolution par qcr\n");
fclose(file_sauv);
qcr(i);
file_sauv=fopen(sauv,"a");
fprintf(file_sauv,"\n");
fprintf(file_sauv,"Resolution par semi 0-1\n");
fclose(file_sauv);
semi_0_1(i);
file_sauv=fopen(sauv,"a");
fprintf(file_sauv,"\n");
fprintf(file_sauv,"Resolution par la linearisation\n");
fclose(file_sauv);
linearisation(i);
}
if(flag_vp)
{
file_sauv=fopen(sauv,"a");
fprintf(file_sauv,"\n");
fprintf(file_sauv,"Resolution par la valeur propre\n");
fclose(file_sauv);
vp(i);
}
if(flag_qcr)

```

```

{
file_sauv=fopen(sauv,"a");
fprintf(file_sauv,"\n");
fprintf(file_sauv,"Resolution par qcr\n");
fclose(file_sauv);
qcr(i);
}
if(flag_semi)
{
file_sauv=fopen(sauv,"a");
fprintf(file_sauv,"\n");
fprintf(file_sauv,"Resolution par semi 0-1\n");
fclose(file_sauv);
semi_0_1(i);
}
if(flag_lin)
{
file_sauv=fopen(sauv,"a");
fprintf(file_sauv,"\n");
fprintf(file_sauv,"Resolution par la linearisation\n");
fclose(file_sauv);
linearisation(i);
}
}
return 1;
}

```

### spectre.sce

```
exec('lambda.sce');  
val=spec(v);  
fprintfMat('spectre.txt',val);  
exit;
```

### resolution\_vp.mos

```
model "resolution vp"  
uses "mmxprs"  
uses "mmquad"  
  
setparam("XPRS_MAXTIME",3600)  
setparam("XPRS_VERBOSE",true)  
setparam("XPRS_MIPLOG",-10000)  
  
declarations  
  N : integer !nombre de variables en 0-1  
  n : integer !nombre de variables entieres  
  m : integer !nombre de contraintes d'egalite  
  p : integer !nombre de contraintes d'inegalite  
end-declarations  
  
initializations from 'donnee_c.dat'  
  N n m p  
end-initializations  
  
declarations  
  VAR=1..N  
  v=1..n  
  M=1..m  
  P=1..p  
  Q: array(VAR,VAR) of integer !Matrice Q modifiee  
  c: array(VAR) of integer      !vecteur c modifie  
  lgc:array(v) of integer       !vecteur des log cumules  
                                !des bornes de x  
  ui:array(v) of integer        !vecteur des bornes de x  
  A: array(M,VAR) of integer    !Matrice A modifiee des  
                                !contraintes d egalite  
  B: array(M) of integer        !vecteur des contraintes  
                                !d'egalite
```

```

D: array(P,VAR) of integer  !Matrice D modifiee des
                             !contraintes d'inegalite
E: array(P) of integer      !vecteur des contraintes
                             !d'inegalite
lambda : real               !plus petite valeur propre

X : array(VAR) of mpvar
end-declarations

forall(i in VAR) X(i) is_binary

initializations from 'donnee_c.dat'
Q c lgc ui A B D E
end-initializations

initializations from 'lambda.dat'
lambda
end-initializations

!objectif
Min2:= (sum(i in VAR)(lambda*(X(i) - X(i)^2)))
Min1 := (sum(i in VAR,j in VAR) Q(i,j)* X(i) *X(j))
Min3:= (sum(i in VAR) c(i) * X(i))
Min:=(Min1 + Min3 +Min2)

!contrainte qui force x à etre <= ui
sum(j in 1..lgc(1))2^(j-1) * X(j) <= ui(1)
forall(i in 2..n) do
  sum(j in (lgc(i-1) + 1)..lgc(i)) 2^(j-1-lgc(i-1)) *
                                     X(j) <= ui(i)
end-do

!contrainte
forall(i in M) do
  sum(j in VAR) A(i,j)*X(j)=B(i)

```

```

end-do
forall(i in P) do
  sum(j in VAR) D(i,j)*X(j) <= E(i)
end-do

minimize(Min)

!ecris le vecteur x dans un fichier
fopen("sol_0_1.txt",F_OUTPUT)
forall(i in VAR)write(" ", getsol(X(i)))
fclose(F_OUTPUT)
end-model

```

## resolution\_qcr.mos

```
model "resolution qcr"
uses "mmxprs"
uses "mmquad"

setparam("XPRS_MAXTIME",3600)
setparam("XPRS_VERBOSE",true)
setparam("XPRS_MIPLOG",-10000)

declarations
  N : integer !nombre de variables en 0-1
  n : integer !nombre de variables entieres
  m : integer !nombre de contraintes d'egalite
  p : integer !nombre de contraintes d'inegalite
end-declarations

initializations from 'donnee_c.dat'
  N n m p
end-initializations

declarations
  VAR=1..N
  v=1..n
  M=1..m
  P=1..p
  Q: array(VAR,VAR) of integer !Matrice Q modifiee
  c: array(VAR) of integer !vecteur c modifie
  lgc:array(v) of integer !vecteur des log cumules
  !des bornes de x
  ui:array(v) of integer !vecteur des bornes de x
  A: array(M,VAR) of integer !Matrice A modifiee des
  !contraintes d egalite
  B: array(M) of integer !vecteur des contraintes
  !d'egalite
```

```
D: array(P,VAR) of integer !Matrice D modifiee des
!contraintes d'inegalite
E: array(P) of integer !vecteur des contraintes
!d'inegalite
u: array(VAR) of real !vecteur de convexification

X : array(VAR) of mpvar
end-declarations

forall(i in VAR) X(i) is_binary

initializations from 'donnee_c.dat'
  Q c lgc ui A B D E
end-initializations

initializations from 'au.dat'
  u
end-initializations

!contrainte
forall(i in M) do
  sum(j in VAR) A(i,j)*X(j) = B(i)
end-do
forall(i in P) do
  sum(j in VAR) D(i,j)*X(j) <= E(i)
end-do

!contrainte qui force x a etre <= ui
sum(j in 1..lgc(1))2^(j-1) * X(j)<= ui(1)
forall(i in 2..n) do
  sum(j in (lgc(i-1) + 1)..lgc(i))2^(j-1-lgc(i-1)) *
  X(j) <= ui(i)
end-do

!objectif
```



```

obj:=(sum(i in VAR,j in VAR) Q(i,j)* X(i) *X(j)) +
      (sum(i in VAR) c(i) * X(i)) +
      (sum(i in VAR)(2*u(i) +0.01)*(X(i)*X(i) - X(i)))

minimize(obj)

!ecris le vecteur x dans un fichier
fopen("sol_0_1.txt",F_OUTPUT)
forall(i in VAR)write(" ", getsol(X(i)))
fclose(F_OUTPUT)
end-model

```

#### resolution\_sem.mos

```

model "semi 0-1"
uses "mmxprs"
uses "mmquad"

setparam("XPRS_MAXTIME",3600)
setparam("XPRS_VERBOSE",true)
setparam("XPRS_MIPLLOG",-10000)

declarations
  N : integer      !nombre de variables entieres
  m : integer      !nombre de contraintes d'egalite
  p : integer      !nombre de contraintes d'inegalite
  maxlg : integer !valeur du max des logs des bornes
end-declarations

initializations from 'donnee_c.dat'
  N m p maxlg
end-initializations

declarations
  VAR=1..N
  M=1..m
  P=1..p
  K=1..maxlg
  Q: array(VAR,VAR) of integer!Matrice Q
  c: array(VAR) of integer   !vecteur c
  lg:array(VAR) of integer   !vecteur des log additionnes
                               !des bornes de x
  ui:array(VAR) of integer   !vecteur des bornes de x
  A: array(M,VAR) of integer !Matrice A modifiee des
                               !contraintes d egalite
  B: array(M) of integer     !vecteur des contraintes
                               !d'egalite

```

```

D: array(P,VAR) of integer !Matrice D modifiee des
                           !contraintes d'inegalite
E: array(P) of integer    !vecteur des contraintes
                           !d'inegalite
lambda : real             !plus petite valeur propre

x : array(VAR) of mpvar
v : array(VAR) of mpvar
t : array(VAR,K) of mpvar
z : array(VAR,K) of mpvar
end-declarations

initializations from 'donnee_c.dat'
Q c lg ui A B D E
end-initializations

initializations from 'lambda.dat'
lambda
end-initializations

!objectif
Min1 := (sum(i in VAR,j in VAR) Q(i,j)* x(i) *x(j))
Min2 := (sum(i in VAR) c(i) *x(i))
Min3 := (sum(i in VAR)-lambda*(x(i)*x(i) - v(i)))
Min := Min1 + Min2 + Min3

!contraintes d'integrites
forall(i in VAR) x(i)is_integer
forall(i in VAR, k in K | k <= lg(i)) t(i,k)is_binary
forall(i in VAR) v(i)is_integer

!contraintes de bornes
forall(i in VAR) x(i) >=0
forall(i in VAR) x(i) <=ui(i)
forall(i in VAR, k in K | k<=lg(i)) z(i,k) >=0

```

```

!contraintes
forall(i in M) do
  sum(j in VAR) A(i,j)*x(j) = B(i)
end-do
forall(i in P) do
  sum(j in VAR) D(i,j)*x(j) <= E(i)
end-do
!contraintes qui forcent v=x^2
forall(i in VAR) x(i) = sum(k in K | k<=lg(i)) (2^(k-1) *
                                                t (i,k))
forall(i in VAR) v(i) = sum(k in K | k<=lg(i))
                        (2^(k-1) * z (i,k))
forall(i in VAR, k in K | k<= lg(i)) z(i,k) <= ui(i)*
                                                t(i,k)
forall(i in VAR, k in K | k<=lg(i)) z(i,k) <= x(i)
forall(i in VAR, k in K | k<=lg(i)) z(i,k) >= x(i) -
                                                ui(i)*(1 - t(i,k))

minimize(Min)

!ecris le vecteur x dans un fichier
fopen("sol.txt",F_OUTPUT)
forall(i in VAR)write(" ", getsol(x(i)))
write("\n")
fclose(F_OUTPUT)
end-model

```

## resolution\_lin.mos

```
model "resolution lin"
  uses "mxxprs"

setparam("XPRS_MAXTIME",3600)
setparam("XPRS_VERBOSE",true)
setparam("XPRS_MIPLLOG",-10000)

declarations
  N : integer      !nombre de variables entieres
  m : integer      !nombre de contraintes d'egalite
  p : integer      !nombre de contraintes d'inegalite
  maxlg : integer !valeur du max des logs des bornes
end-declarations

initializations from 'donnee_c.dat'
  N m p maxlg
end-initializations

declarations
  VAR=1..N
  M=1..m
  P=1..p
  K=1..maxlg
  Q: array(VAR,VAR) of integer!Matrice Q
  c: array(VAR) of integer      !vecteur c
  lg:array(VAR) of integer      !vecteur des log additionnes
                                !des bornes de x
  ui:array(VAR) of integer      !vecteur des bornes de x
  A: array(M,VAR) of integer    !Matrice A modifiee des
                                !contraintes d egalite
  B: array(M) of integer        !vecteur des contraintes
                                !d'egalite
  D: array(P,VAR) of integer    !Matrice D modifiee des
```

```
                                !contraintes d'inegalite
  E: array(P) of integer        !vecteur des contraintes
                                !d'inegalite

  x : array(VAR) of mpvar
  t : array(VAR,K) of mpvar
  y : array(VAR,VAR) of mpvar
  z : array(VAR,VAR,K) of mpvar

end-declarations

initializations from 'donnee_c.dat'
  Q c lg ui A B D E
end-initializations

!objectif
  Min1 := (sum(i in VAR,j in VAR | i<=j) Q(i,j)* y(i,j))
  Min2 := (sum(i in VAR,j in VAR | i>j)Q(i,j)* y(j,i))
  Min3 := (sum(i in VAR) c(i) * x(i))
  Min:= Min1 + Min2 + Min3

!contraintes d'integrites
  forall(i in VAR) x(i) is_integer
  forall(i in VAR,k in K | k<=lg(i)) t(i,k) is_binary
  forall(i in VAR, j in VAR | i <= j) y(i,j) is_integer
  forall(i in VAR, j in VAR | i <= j )
  forall(k in K | k<=lg(i)) z(i,j,k) is_integer

!contraintes de bornes
  forall(i in VAR) x(i) >=0
  forall(i in VAR) x(i) <=ui(i)
  forall(i in VAR, j in VAR | i<= j) y(i,j) >=0
  forall(i in VAR, j in VAR | i <= j )
                                forall(k in K | k<=lg(i)) z(i,j,k) >=0
```

```

!contraintes
forall(i in M) do
  sum(j in VAR) A(i,j)*x(j) = B(i)
end-do
forall(i in P) do
  sum(j in VAR) D(i,j)*x(j) <= E(i)
end-do

!contraintes qui forcent y(i,j)=x(i)*x(j)
forall(i in VAR) x(i) = sum(k in K | k<=lg(i)) (2^(k-1) *
                                                t(i,k))

forall(i in VAR, j in VAR | i <= j) y(i,j) =
  sum(k in K | k<=lg(i)) (2^(k-1)*z(i,j,k))
forall(i in VAR, j in VAR | i <= j )
  forall(k in K | k<=lg(i)) z(i,j,k) <= ui(j)*t(i,k)
forall(i in VAR, j in VAR | i <= j )
  forall(k in K | k<=lg(i)) z(i,j,k) <= x(j)
forall(i in VAR, j in VAR | i <= j )
  forall(k in K | k<=lg(i)) z(i,j,k) >= x(j) - ui(j)*
  (1 - t(i,k))

minimize(Min)

!ecris le vecteur x dans un fichier
fopen("sol.txt",F_OUTPUT)
forall(i in VAR)write(" ", getsol(x(i)))
fclose(F_OUTPUT)
end-model

```