

ÉCOLE NORMALE SUPÉRIEURE PARIS-SACLAY
CENTRE DE MATHÉMATIQUES ET DE LEURS APPLICATIONS

RAPPORT DE STAGE DE LICENCE

Bundle adjustment with known positions

Antoine BARRIER
Ségolène MARTIN

Encadrants : Carlo DE FRANCHIS
Jean-Michel MOREL

25 Janvier 2017 — 29 Juin 2017

Introduction

Contexte et situation

La modélisation en 3 dimensions à partir d'images réalisées par une ou plusieurs caméra(s) est un problème dont les applications se multiplient avec le développement de nouvelles technologies : cartographie 3D, estimation des dégâts lors d'une catastrophe naturelle, estimation de la fonte des glaciers ...

La technique de modélisation 3D la plus couramment employée s'appuie sur le principe de la **vision stéréoscopique** : les relations entre deux photographies d'une même scène rigide prises depuis deux points de vue différents permettent de reconstituer le relief de la scène.

Le problème du Bundle Adjustment

Cependant, pour calculer les coordonnées 3D d'un point identifié sur plusieurs images, il est nécessaire de connaître les **positions** et les **orientations** des caméras. Ce n'est en général pas le cas, ou alors à une incertitude près qui ne permet pas d'avoir une reconstitution satisfaisante.

[**On essaye donc d'estimer positions et coordonnées à partir de correspondances entre les images. Cette méthode, connue sous le nom de bundle adjustment, permet en théorie d'estimer une reconstruction 3D de la scène modulo une similitude de l'espace.**] Pour palier à ce problème, il existe une méthode connue sous le nom de Bundle Adjustment, qui fournit une réestimation de chacun des paramètres (coordonnées des points dans les images, orientations et position des caméras) de sorte à ce que ces derniers donnent une reconstitution 3D satisfaisante. Cependant, les algorithmes d'optimisation actuels utilisés dans le Bundle Adjustment ne semblent pas fonctionner totalement : s'ils permettent bien de remodéliser la scène 3D, c'est en fait plus du à une compensation d'erreurs qu'à une estimation exacte des paramètres de la caméra. Nous cherchons donc à estimer de manière exacte ces paramètres.

[**Pour remédier à cela,**] nous avons travaillé sur un cas particulier du problème : celui où les positions des caméras sont connues. Cette hypothèse permet de simplifier la phase de bundle adjustment (moins de paramètres sont réestimés) et semble de plus amplement justifiée : de nombreux systèmes (voitures, drones, robots, avions) embarquent des caméras munies d'un système type GPS qui permet de connaître les positions des caméras avec une précision suffisante (de l'ordre du cm). C'est alors la mesure de l'orientation qui est la plus sensible, car suivant la distance entre la caméra et la scène, une erreur angulaire même très faible peut induire une erreur importante sur la position des points 3D (un milli-radian à 100 mètres de distance donne une erreur de 10 cm).

Cette situation semble aussi particulièrement adaptée aux images satellites, les positions des caméras étant connues avec une forte précision puisque les trajectoires sont programmées à l'avance. Résoudre ce problème pourrait donc permettre une cartographie 3D précise de la Terre.

Objectifs

L'objectif de ce stage est d'étudier le problème du bundle adjustment dans le cas où les positions des caméras sont supposées connues précisément. Le but étant de **trouver le nombre minimal de vues nécessaires pour obtenir une reconstruction 3D de la scène à partir de mesures faites uniquement sur les images** (c'est-à-dire des correspondances entre images obtenues de façon entièrement automatique avec des algorithmes comme la méthode SIFT dont la précision est de l'ordre de 0.1 pixel) [**et les angles des caméras**].

On pourra alors programmer un algorithme capable de retourner les paramètres des caméras associées à différentes prises de vues avec une précision contrôlée en ne connaissant qu'une estimation de ces paramètres ainsi que des points de concordance entre images.

Travail

Au cours de ce stage, nous avons eu différentes tâches à réaliser :

- un travail bibliographique pour assimiler les connaissances sur le sujet,
- une identification des difficultés de notre problème, mais aussi des simplifications apportées par les hypothèses dans lesquelles nous nous sommes placés,
- une implémentation en Python de nos méthodes, mais aussi des méthodes de bundle adjustment classique, afin de les comparer,
- tester notre algorithme sur des données simulées puis sur des données réelles.
- [la publication d'un article ???]

Remerciements

Remerciements à Jean-Michel MOREL et à Carlo DE FRANCHIS pour l'aide apportée.

Plan du rapport

1 Description du problème	4
1.1 Rappels sur les caméras	4
1.2 Situation et notations	4
1.3 Démarche	5
2 Formules associées au problème	6
2.1 Deux caméras et un point	6
2.2 Généralisation	8
3 Étude des matrices jacobiniennes partielles A et B	10
3.1 Étude de la matrice A	10
3.2 Étude de la matrice B	10
4 Simulations	12
4.1 Le cas satellitaire	12
4.2 Un cas terrestre	19
4.3 Améliorations de la méthode : ajout d'un terme forçant	25

5 Application à des images satellitaires	29
5.1 Implémentation de notre méthode	29
5.2 Comparaison avec les méthodes classiques de Bundle Adjustment	32
A Fonctions de base	35
B Simulations sans données réelles	43

1 Description du problème

1.1 Rappels sur les caméras

On modélise une caméra par la donnée d'une matrice de taille 3×4 de la forme $P = KR[I_3 | -C]$ où :

- K contient les paramètres internes de la caméra : distance focale, résolution, informations sur le repère de l'image ...,
- R modélise l'orientation de la caméra : c'est une matrice de rotation,
- C est la position du centre de la caméra.

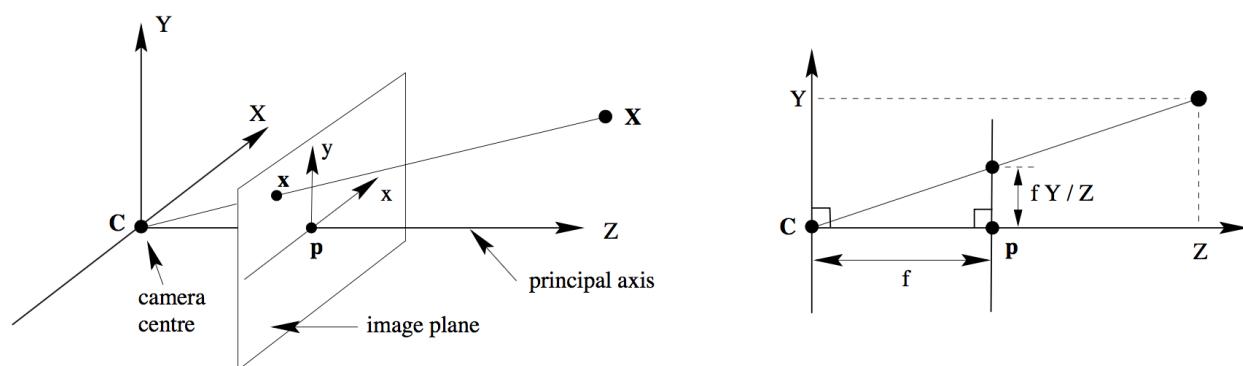


Figure 1: Modélisation d'une caméra [changer notations figure ...]

Pour des descriptions plus détaillées, on pourra se référer au chapitre 6 du livre de HARTLEY et ZISSEMAN [1].

1.2 Situation et notations

On se place dans le référentiel terrestre noté $\mathbf{R}_T = (O, X_T, Y_T, Z_T)$.

On s'intéresse à K images prises par des caméras à K positions distinctes **connues précisément** pour lesquelles on a N points de correspondance (grâce à la méthode SIFT ...).

On note $\mathbf{R}_{cam\ i} = (C_i, X_{cam\ i}, Y_{cam\ i}, Z_{cam\ i})$ le repère de la i -ième caméra, $\mathbf{R}_{im\ i}$ le repère du plan image de la i -ième caméra et on suppose précisément connues les coordonnées du centre C_i de chaque caméra.

On suppose de plus que les matrices K_i des paramètres internes des caméras sont toutes égales à

$$K = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Pour des questions de compatibilité avec nos codes Python, on numérottera les caméras de 0 à $K - 1$ et les points de 0 à $N - 1$.

On dispose également :

- d'une **estimation** – pas assez précise – des angles de rotation des caméras, que l'on regroupe selon le vecteur :

$$\theta_0 = (\tilde{\alpha}_0, \tilde{\beta}_0, \tilde{\gamma}_0, \dots, \tilde{\alpha}_{K-1}, \tilde{\beta}_{K-1}, \tilde{\gamma}_{K-1})^\top$$

où $\tilde{\alpha}_i$, $\tilde{\beta}_i$ et $\tilde{\gamma}_i$ sont les angles de rotation estimés de la i -ième caméra respectivement par rapport à X_T , Y_T et Z_T . Ainsi, la matrice de rotation de la i -ième caméra se décompose sous la forme :

$$R_i = \begin{pmatrix} \cos(\tilde{\gamma}_i) & -\sin(\tilde{\gamma}_i) & 0 \\ \sin(\tilde{\gamma}_i) & \cos(\tilde{\gamma}_i) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\tilde{\beta}_i) & 0 & \sin(\tilde{\beta}_i) \\ 0 & 1 & 0 \\ -\sin(\tilde{\beta}_i) & 0 & \cos(\tilde{\beta}_i) \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\tilde{\alpha}_i) & -\sin(\tilde{\alpha}_i) \\ 0 & \sin(\tilde{\alpha}_i) & \cos(\tilde{\alpha}_i) \end{pmatrix}$$

Remarque L'ordre des matrices et donc des trois rotations sous-entend que l'on considère des rotations définies de manière extrinsèque.

- d'une **estimation** – là encore pas assez précise¹ – des coordonnées des images sur chaque caméra des points pour lesquels on a une correspondance, regroupés selon le vecteur :

$$\mathbf{X}_0 = (\underbrace{\tilde{x}_0^0, \tilde{y}_0^0, \dots, \tilde{x}_i^0, \tilde{y}_i^0, \dots, \tilde{x}_{K-1}^0, \tilde{y}_{K-1}^0, \dots, \tilde{x}_0^j, \tilde{y}_0^j, \dots, \tilde{x}_{K-1}^j, \tilde{y}_{K-1}^j, \dots, \tilde{x}_0^{N-1}, \tilde{y}_0^{N-1}, \dots, \tilde{x}_{K-1}^{N-1}, \tilde{y}_{K-1}^{N-1}}_{\text{coord. des images du point 0}}, \underbrace{\tilde{x}_0^j, \tilde{y}_0^j, \dots, \tilde{x}_{K-1}^j, \tilde{y}_{K-1}^j}_{\text{coord. des images du point } j}, \underbrace{\tilde{x}_0^{N-1}, \tilde{y}_0^{N-1}, \dots, \tilde{x}_{K-1}^{N-1}, \tilde{y}_{K-1}^{N-1}}_{\text{coord. des images du point } N-1})^\top$$

où $(\tilde{x}_i^j, \tilde{y}_i^j)^\top$ sont les coordonnées estimées de l'image du j -ième point sur la i -ième caméra (donc exprimées dans le repère $R_{im\ i}$).

1.3 Démarche

Notre problème consiste à estimer avec la meilleure précision possible les valeurs réelles θ_* des angles ainsi que \mathbf{X}_* des coordonnées des points que l'on regroupera dans des vecteurs.

Pour cela, on dispose des valeurs de θ_0 et de \mathbf{X}_0 ainsi que des conditions imposées par les correspondances des points, à savoir le fait que pour tout $0 \leq j \leq N-1$ fixé, on sait que les droites $(C_i \mathbf{x}_i^j)$ sont toutes sécantes en un même point (qui est le j -ième point de correspondance mais dont on ne connaît pas les coordonnées).

Ces conditions vont nous permettre de voir notre estimation (θ_a, \mathbf{X}_a) de (θ_*, \mathbf{X}_*) comme la solution d'une équation du type $F(\theta, \mathbf{X}) = 0$ au voisinage de (θ_0, \mathbf{X}_0) .

¹les algorithmes de correspondance n'ont pas une précision parfaite

2 Formules associées au problème

2.1 Deux caméras et un point

On considère ici le cas où $K = 2$ et $N = 1$.²

On a $\theta_0 = (\tilde{\alpha}_1, \tilde{\beta}_1, \tilde{\gamma}_1, \tilde{\alpha}_2, \tilde{\beta}_2, \tilde{\gamma}_2)^\top$ et $\mathbf{X}_0 = (\tilde{x}_1, \tilde{y}_1, \tilde{x}_2, \tilde{y}_2)^\top$.

Notons $\theta_a = (\alpha_1, \beta_1, \gamma_1, \alpha_2, \beta_2, \gamma_2)^\top$ et $\mathbf{X}_a = (x_1, y_1, x_2, y_2)^\top$ les angles et positions que l'on va estimer.

On sait que les droites de projection du point \mathbf{X} sur les deux caméras sont sécantes (en \mathbf{X}). On obtient alors facilement l'existence de λ et μ tels que³ :

$$\tilde{C}_1 + \lambda R_1^{-1} K^{-1} \mathbf{x}_1 = \tilde{C}_2 + \mu R_2^{-1} K^{-1} \mathbf{x}_2 (= \tilde{\mathbf{X}})$$

Ainsi les trois vecteurs $\tilde{C}_2 - \tilde{C}_1$, $R_2^{-1} K^{-1} \mathbf{x}_2$ et $R_1^{-1} K^{-1} \mathbf{x}_1$ sont liés, ce qui se traduit par le fait que :

$$\det(R_1^{-1} K^{-1} \mathbf{x}_1 | R_2^{-1} K^{-1} \mathbf{x}_2 | \tilde{C}_2 - \tilde{C}_1) = 0 \quad (1)$$

Calculons les valeurs de $R_1^{-1} K^{-1} \mathbf{x}_1$ et $R_2^{-1} K^{-1} \mathbf{x}_2$:

Pour simplifier les expressions on notera c_θ pour $\cos(\theta)$ et s_θ pour $\sin(\theta)$. On a :

$$\begin{aligned} R_i^{-1} K^{-1} \mathbf{x}_i &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & c_{\alpha_i} & s_{\alpha_i} \\ 0 & -s_{\alpha_i} & c_{\alpha_i} \end{pmatrix} \begin{pmatrix} c_{\beta_i} & 0 & -s_{\beta_i} \\ 0 & 1 & 0 \\ s_{\beta_i} & 0 & c_{\beta_i} \end{pmatrix} \begin{pmatrix} c_{\gamma_i} & s_{\gamma_i} & 0 \\ -s_{\gamma_i} & c_{\gamma_i} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1/f & 0 & 0 \\ 0 & 1/f & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & c_{\alpha_i} & s_{\alpha_i} \\ 0 & -s_{\alpha_i} & c_{\alpha_i} \end{pmatrix} \begin{pmatrix} c_{\beta_i} & 0 & -s_{\beta_i} \\ 0 & 1 & 0 \\ s_{\beta_i} & 0 & c_{\beta_i} \end{pmatrix} \begin{pmatrix} c_{\gamma_i} & s_{\gamma_i} & 0 \\ -s_{\gamma_i} & c_{\gamma_i} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_i/f \\ y_i/f \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & c_{\alpha_i} & s_{\alpha_i} \\ 0 & -s_{\alpha_i} & c_{\alpha_i} \end{pmatrix} \begin{pmatrix} c_{\beta_i} & 0 & -s_{\beta_i} \\ 0 & 1 & 0 \\ s_{\beta_i} & 0 & c_{\beta_i} \end{pmatrix} \begin{pmatrix} c_{\gamma_i} x_i/f + s_{\gamma_i} y_i/f \\ -s_{\gamma_i} x_i/f + c_{\gamma_i} y_i/f \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & c_{\alpha_i} & s_{\alpha_i} \\ 0 & -s_{\alpha_i} & c_{\alpha_i} \end{pmatrix} \begin{pmatrix} c_{\beta_i} (c_{\gamma_i} x_i/f + s_{\gamma_i} y_i/f) - s_{\beta_i} \\ -s_{\gamma_i} x_i/f + c_{\gamma_i} y_i/f \\ s_{\beta_i} (c_{\gamma_i} x_i/f + s_{\gamma_i} y_i/f) + c_{\beta_i} \end{pmatrix} \\ &= \begin{pmatrix} c_{\beta_i} (c_{\gamma_i} x_i/f + s_{\gamma_i} y_i/f) - s_{\beta_i} \\ c_{\alpha_i} (-s_{\gamma_i} x_i/f + c_{\gamma_i} y_i/f) + s_{\alpha_i} [s_{\beta_i} (c_{\gamma_i} x_i/f + s_{\gamma_i} y_i/f) + c_{\beta_i}] \\ s_{\alpha_i} (s_{\gamma_i} x_i/f - c_{\gamma_i} y_i/f) + c_{\alpha_i} [s_{\beta_i} (c_{\gamma_i} x_i/f + s_{\gamma_i} y_i/f) + c_{\beta_i}] \end{pmatrix} \end{aligned}$$

Nous verrons par la suite que l'on a surtout besoin de calculer les dérivées partielles du déterminant (1) par rapport aux variables $\alpha_i, \beta_i, \gamma_i, x_i$ et y_i .

$$\text{Notons } \det(R_1^{-1} K^{-1} \mathbf{x}_1 | R_2^{-1} K^{-1} \mathbf{x}_2 | \tilde{C}_2 - \tilde{C}_1) = \det \begin{pmatrix} k^1 & l^1 & m^1 \\ k^2 & l^2 & m^2 \\ k^3 & l^3 & m^3 \end{pmatrix}.$$

²exceptionnellement on numérottera les caméras 1 et 2 et non 0 et 1, cela étant plus parlant dans une section théorique sans application directe en Python

³les ~ indiquent qu'on utilise des coordonnées non homogènes

Considérons alors les mineurs d'ordre 2 des deux premières colonnes et calculons les dérivées partielles⁴ :

- de $f^2 \det \begin{pmatrix} k^1 & l^1 \\ k^2 & l^2 \end{pmatrix}$:

On rappelle que l'on a :

- $[fk^1] = c_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) - fs_{\beta_1}$
- $[fk^2] = c_{\alpha_1}(-s_{\gamma_1}x_1 + c_{\gamma_1}y_1) + s_{\alpha_1}[s_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) + fc_{\beta_1}]$
- $[fl^1] = c_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) - fs_{\beta_2}$
- $[fl^2] = c_{\alpha_2}(-s_{\gamma_2}x_2 + c_{\gamma_2}y_2) + s_{\alpha_2}[s_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) + fc_{\beta_2}]$

On obtient :

Variable	Dérivée partielle
α_1	$-[fl^1](-s_{\alpha_1}(-s_{\gamma_1}x_1 + c_{\gamma_1}y_1) + c_{\alpha_1}[s_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) + fc_{\beta_1}])$
α_2	$[fk^1](-s_{\alpha_2}(-s_{\gamma_2}x_2 + c_{\gamma_2}y_2) + c_{\alpha_2}[s_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) + fc_{\beta_2}])$
β_1	$(-s_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) - fc_{\beta_1})[fl^2] - [fl^1]s_{\alpha_1}[c_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) - fs_{\beta_1}]$
β_2	$[fk^1]s_{\alpha_2}[c_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) - fs_{\beta_2}] - (-s_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) - fc_{\beta_2})[fk^2]$
γ_1	$c_{\beta_1}(-s_{\gamma_1}x_1 + c_{\gamma_1}y_1)[fl^2] - [fl^1](c_{\alpha_1}(-c_{\gamma_1}x_1 - s_{\gamma_1}y_1) + s_{\alpha_1}s_{\beta_1}(-s_{\gamma_1}x_1 + c_{\gamma_1}y_1))$
γ_2	$[fk^1](c_{\alpha_2}(-c_{\gamma_2}x_2 - s_{\gamma_2}y_2) + s_{\alpha_2}s_{\beta_2}(-s_{\gamma_2}x_2 + c_{\gamma_2}y_2)) - c_{\beta_2}(-s_{\gamma_2}x_2 + c_{\gamma_2}y_2)[fk^2]$
x_1	$c_{\beta_1}c_{\gamma_1}[fl^2] - [fl^1](-c_{\alpha_1}s_{\gamma_1} + s_{\alpha_1}s_{\beta_1}c_{\gamma_1})$
x_2	$[fk^1](-c_{\alpha_2}s_{\gamma_2} + s_{\alpha_2}s_{\beta_2}c_{\gamma_2}) - c_{\beta_2}c_{\gamma_2}[fk^2]$
y_1	$c_{\beta_1}s_{\gamma_1}[fl^2] - [fl^1](c_{\alpha_1}c_{\gamma_1} + s_{\alpha_1}s_{\beta_1}s_{\gamma_1})$
y_2	$[fk^1](c_{\alpha_2}c_{\gamma_2} + s_{\alpha_2}s_{\beta_2}s_{\gamma_2}) - c_{\beta_2}s_{\gamma_2}[fk^2]$

- de $f^2 \det \begin{pmatrix} k^1 & l^1 \\ k^3 & l^3 \end{pmatrix}$:

On rappelle que l'on a :

- $[fk^1] = c_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) - fs_{\beta_1}$
- $[fk^3] = s_{\alpha_1}(s_{\gamma_1}x_1 - c_{\gamma_1}y_1) + c_{\alpha_1}[s_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) + fc_{\beta_1}]$
- $[fl^1] = c_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) - fs_{\beta_2}$
- $[fl^3] = s_{\alpha_2}(s_{\gamma_2}x_2 - c_{\gamma_2}y_2) + c_{\alpha_2}[s_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) + fc_{\beta_2}]$

On obtient :

Variable	Dérivée partielle
α_1	$-[fl^1](c_{\alpha_1}(s_{\gamma_1}x_1 - c_{\gamma_1}y_1) - s_{\alpha_1}[s_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) + fc_{\beta_1}])$
α_2	$[fk^1](c_{\alpha_2}(s_{\gamma_2}x_2 - c_{\gamma_2}y_2) - s_{\alpha_2}[s_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) + fc_{\beta_2}])$
β_1	$(-s_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) - fc_{\beta_1})[fl^3] - [fl^1]c_{\alpha_1}[c_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) - fs_{\beta_1}]$
β_2	$[fk^1]c_{\alpha_2}[c_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) - fs_{\beta_2}] - (-s_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) - fc_{\beta_2})[fk^3]$
γ_1	$c_{\beta_1}(-s_{\gamma_1}x_1 + c_{\gamma_1}y_1)[fl^3] - [fl^1](s_{\alpha_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) + c_{\alpha_1}[s_{\beta_1}(-s_{\gamma_1}x_1 + c_{\gamma_1}y_1)])$
γ_2	$[fk^1](s_{\alpha_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) + c_{\alpha_2}[s_{\beta_2}(-s_{\gamma_2}x_2 + c_{\gamma_2}y_2)]) - c_{\beta_2}(-s_{\gamma_2}x_2 + c_{\gamma_2}y_2)[fk^3]$
x_1	$c_{\beta_1}c_{\gamma_1}[fl^3] - [fl^1](s_{\alpha_1}s_{\gamma_1} + c_{\alpha_1}s_{\beta_1}c_{\gamma_1})$
x_2	$[fk^1](s_{\alpha_2}s_{\gamma_2} + c_{\alpha_2}s_{\beta_2}c_{\gamma_2}) - c_{\beta_2}c_{\gamma_2}[fk^3]$
y_1	$c_{\beta_1}s_{\gamma_1}[fl^3] - [fl^1](-s_{\alpha_1}c_{\gamma_1} + c_{\alpha_1}s_{\beta_1}s_{\gamma_1})$
y_2	$[fk^1](-s_{\alpha_2}c_{\gamma_2} + c_{\alpha_2}s_{\beta_2}s_{\gamma_2}) - c_{\beta_2}s_{\gamma_2}[fk^3]$

⁴on multiplie par f^2 pour simplifier les formules, cela étant sans importance

- de $f^2 \det \begin{pmatrix} k^2 & l^2 \\ k^3 & l^3 \end{pmatrix}$:

On rappelle que l'on a :

- $[fk^2] = c_{\alpha_1}(-s_{\gamma_1}x_1 + c_{\gamma_1}y_1) + s_{\alpha_1}[s_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) + fc_{\beta_1}]$
- $[fk^3] = s_{\alpha_1}(s_{\gamma_1}x_1 - c_{\gamma_1}y_1) + c_{\alpha_1}[s_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) + fc_{\beta_1}]$
- $[fl^2] = c_{\alpha_2}(-s_{\gamma_2}x_2 + c_{\gamma_2}y_2) + s_{\alpha_2}[s_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) + fc_{\beta_2}]$
- $[fl^3] = s_{\alpha_2}(s_{\gamma_2}x_2 - c_{\gamma_2}y_2) + c_{\alpha_2}[s_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) + fc_{\beta_2}]$

On obtient :

Variable	Dérivée partielle
α_1	$(-s_{\alpha_1}(-s_{\gamma_1}x_1 + c_{\gamma_1}y_1) + c_{\alpha_1}[s_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) + fc_{\beta_1}])[fl^3]$ - $[fl^2](c_{\alpha_1}(s_{\gamma_1}x_1 - c_{\gamma_1}y_1) - s_{\alpha_1}[s_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1)])$
α_2	$[fk^2](c_{\alpha_2}(s_{\gamma_2}x_2 - c_{\gamma_2}y_2) - s_{\alpha_2}[s_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) + fc_{\beta_2}])$ - $(-s_{\alpha_2}(-s_{\gamma_2}x_2 + c_{\gamma_2}y_2) + c_{\alpha_2}[s_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) + fc_{\beta_2})][fk^3]$
β_1	$s_{\alpha_1}[c_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) - fs_{\beta_1}][fl^3]$ - $[fl^2]c_{\alpha_1}[c_{\beta_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) - fs_{\beta_1}]$
β_2	$[fk^2]c_{\alpha_2}[c_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) - fs_{\beta_2}]$ - $(s_{\alpha_2}[c_{\beta_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) - fs_{\beta_2})][fk^3]$
γ_1	$(c_{\alpha_1}(-c_{\gamma_1}x_1 - s_{\gamma_1}y_1) + s_{\alpha_1}s_{\beta_1}(-s_{\gamma_1}x_1 + c_{\gamma_1}y_1))[fl^3]$ - $[fl^2](s_{\alpha_1}(c_{\gamma_1}x_1 + s_{\gamma_1}y_1) + c_{\alpha_1}s_{\beta_1}(-s_{\gamma_1}x_1 + c_{\gamma_1}y_1))$
γ_2	$[fk^2](s_{\alpha_2}(c_{\gamma_2}x_2 + s_{\gamma_2}y_2) + c_{\alpha_2}s_{\beta_2}(s_{\gamma_2}x_2 - c_{\gamma_2}y_2))$ - $(c_{\alpha_2}(-c_{\gamma_2}x_2 - s_{\gamma_2}y_2) + s_{\alpha_2}s_{\beta_2}(-s_{\gamma_2}x_2 + c_{\gamma_2}y_2))[fk^3]$
x_1	$(-c_{\alpha_1}s_{\gamma_1} + s_{\alpha_1}s_{\beta_1}c_{\gamma_1})[fl^3] - [fl^2](s_{\alpha_1}s_{\gamma_1} + c_{\alpha_1}s_{\beta_1}c_{\gamma_1})$
x_2	$[fk^2](s_{\alpha_2}s_{\gamma_2} + c_{\alpha_2}s_{\beta_2}c_{\gamma_2}) - (-c_{\alpha_2}s_{\gamma_2} + s_{\alpha_2}s_{\beta_2}c_{\gamma_2})[fk^3]$
y_1	$(c_{\alpha_1}c_{\gamma_1} + s_{\alpha_1}s_{\gamma_1})[fl^3] - [fl^2](-s_{\alpha_1}c_{\gamma_1} + c_{\alpha_1}s_{\beta_1}s_{\gamma_1})$
y_2	$[fk^2](-s_{\alpha_2}c_{\gamma_2} + c_{\alpha_2}s_{\beta_2}s_{\gamma_2}) - (c_{\alpha_2}c_{\gamma_2}y_2 + s_{\alpha_2}s_{\beta_2}s_{\gamma_2})[fk^3]$

On utilise alors la formule :

$$\det(R_1^{-1}K^{-1}\mathbf{x}_1 | R_2^{-1}K^{-1}\mathbf{x}_2 | \tilde{\mathbf{C}}_2 - \tilde{\mathbf{C}}_1) = m_1(k_2l_3 - k_3l_2) - m_2(k_1l_3 - k_3l_1) + m_3(k_1l_2 - k_2l_1)$$

où $m_1 = x_{C_2} - x_{C_1}$, $m_2 = y_{C_2} - y_{C_1}$ et $m_3 = z_{C_2} - z_{C_1}$

On obtient alors les dérivées partielles par rapport à chacune des variables de notre déterminant en prenant les combinaisons linéaires des trois tableaux précédents.

2.2 Généralisation

Lorsque l'on a plusieurs caméras et plusieurs points, on doit chercher (θ_a, \mathbf{X}_a) qui annulent tous les déterminants entre deux matrices et un point.

Posons alors $F : \mathbb{R}^{3K} \times \mathbb{R}^{2NK} \rightarrow \mathbb{R}^{\binom{K}{2}N}$ où \det_{i_1, i_2}^j est le déterminant de la forme (1) associé aux caméras i_1, i_2 et au point j .

Notre problème consiste donc à chercher (θ_a, \mathbf{X}_a) qui annulent F au voisinage de (θ_0, \mathbf{X}_0) .

2.2.1 Stratégie

On applique une formule de TAYLOR à l'ordre 1 :

$$F(\theta, \mathbf{X}) = F(\theta_0, \mathbf{X}_0) + dF_\theta(\theta_0, \mathbf{X}_0)d\theta + dF_{\mathbf{X}}(\theta_0, \mathbf{X}_0)d\mathbf{X} + o((d\theta, d\mathbf{X}))$$

où $d\theta = \theta - \theta_0$ et $d\mathbf{X} = \mathbf{X} - \mathbf{X}_0$.

On résout alors $F(\theta, \mathbf{X}) = 0$ au premier ordre (l'erreur étant supposée faible), c'est-à-dire on cherche $(d\theta, d\mathbf{X})$ tels que :

$$\left(\frac{\partial F}{\partial \theta}(\theta_0, \mathbf{X}_0), \frac{\partial F}{\partial \mathbf{X}}(\theta_0, \mathbf{X}_0) \right) \begin{pmatrix} d\theta \\ d\mathbf{X} \end{pmatrix} = -F(\theta_0, \mathbf{X}_0)$$

Dans la suite, on notera $A := \frac{\partial F}{\partial \theta}(\theta_0, \mathbf{X}_0)$ et $B := \frac{\partial F}{\partial \mathbf{X}}(\theta_0, \mathbf{X}_0)$.

Notre problème consiste donc à inverser la matrice $M := (A, B)$. Cependant, comme on va avoir beaucoup de points de correspondance, on aura beaucoup plus d'équations que d'inconnues (le système est sur-déterminé), et donc notre matrice ne sera pas inversible. On espère donc que les équations à première vue incompatibles seront globalement redondantes et nous permettront ainsi de gagner en précision. On va donc calculer la pseudo-inverse de M que l'on notera M^+ .

La matrice $-M^+F(\theta_0, \mathbf{X}_0)$ fournit alors la meilleure approximation de $\begin{pmatrix} d\theta \\ d\mathbf{X} \end{pmatrix}$ au sens des moindres carrés.

On peut montrer que dans notre cas :

$$M^+ = (M^\top M)^{-1}M^\top$$

Algorithmiquement, la pseudo-inverse s'obtient à partir de la décomposition en valeurs singulières de M . On décompose $M = U\Sigma V^\top$ puis on calcule $M^+ = V\Sigma^+U^\top$, où Σ^+ est l'inverse de la matrice diagonale Σ et s'obtient donc aisément.

Remarque Des bibliothèques Python permettront de calculer les pseudo-inverses aisément.

2.2.2 Condition nécessaire sur le nombre de caméras

Pour calculer la pseudo inverse de $M = (A, B)$, on doit avoir plus d'équations que d'inconnues, c'est-à-dire que M doit avoir plus de lignes que de colonnes.

On rappelle que M possède $\binom{K}{2}N$ lignes et $3K + 2NK$ colonnes.

On résout donc $\binom{K}{2}N \geq (3 + 2N)K$, et on obtient :

$$N(K - 5) \geq 6$$

Il faut donc au moins 6 caméras pour espérer calculer la pseudo inverse.

3 Étude des matrices jacobniennes partielles A et B

3.1 Étude de la matrice A

A est la matrice jacobienne $D_\theta F \in \mathcal{M}_{\binom{K}{2}N, 3K}$. On s'intéresse donc aux dérivées de F par rapport à tous les angles.

On remarque que les \det_{i_1, i_2}^j ne dépendent que des angles $\alpha_{i_1}, \beta_{i_1}, \gamma_{i_1}, \alpha_{i_2}, \beta_{i_2}$ et γ_{i_2} , et donc leurs dérivées par rapport à tous les angles associés aux caméras i telles que $i \neq i_1$ et $i \neq i_2$ sont nulles. Ainsi, la matrice A possède de nombreux 0. Essayons de comprendre la structure de A .

Pour commencer, choisissons une convention pour l'ordre des composantes de F (lignes de A) et des angles de θ (colonnes de A).

- Concernant les lignes, on associera aux lignes l comprises entre $\binom{K}{2}j \leq l < \binom{K}{2}(j+1)$ les déterminants associés au j -ième point de correspondance ($0 \leq j \leq N-1$), c'est-à-dire de la forme \det_{i_1, i_2}^j . Reste à choisir comment organiser au sein de ces groupes de lignes l'ordre des indices (i_1, i_2) . On écrit les lignes dans l'ordre $(0, 1), (0, 2), \dots, (0, K-1), (1, 2), (1, 3), \dots, (1, K-1), \dots, (K-2, K-1)$.
- Concernant les colonnes, on dérivera pour A les composantes par rapport aux variables suivant l'ordre $\underbrace{(\alpha_0, \beta_0, \gamma_0, \alpha_1, \beta_1, \gamma_1, \dots, \alpha_{K-1}, \beta_{K-1}, \gamma_{K-1})}_{\theta^\top}$.

Exemple dans le cas $K = 4$: La matrice A est de la forme :

$$A = \begin{pmatrix} A_1 \\ \vdots \\ A_j \\ \vdots \\ A_N \end{pmatrix} \text{ où } A_j = \begin{array}{c|ccc|ccc|ccc|ccc|c} l & \alpha_0 & \beta_0 & \gamma_0 & \alpha_1 & \beta_1 & \gamma_1 & \alpha_2 & \beta_2 & \gamma_2 & \alpha_3 & \beta_3 & \gamma_3 & (i_1, i_2) \\ \hline 6j & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (0, 1) \\ 6j+1 & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (0, 2) \\ 6j+2 & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (0, 3) \\ 6j+3 & 0 & 0 & 0 & & & & & & & 0 & 0 & 0 & (1, 2) \\ 6j+4 & 0 & 0 & 0 & & & & & & & 0 & 0 & 0 & (1, 3) \\ 6j+5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (2, 3) \end{array}$$

Dans le cas général, A est composé des matrices A_j de taille $\binom{K}{2} \times 3K$.

Ensuite, les coefficients non nuls de A se calculent en utilisant les formules de la section précédente.

3.2 Étude de la matrice B

Commençons, comme pour A , par choisir une convention pour l'ordre des composantes de F (lignes de B) et des perturbations de δ (colonnes de B) :

- Concernant les lignes, on garde la même convention que pour A ,
- Concernant les colonnes, on dérivera les composantes par rapport aux variables suivant l'ordre $\underbrace{(\mathbf{x}_0^0, \mathbf{y}_0^0, \dots, \mathbf{x}_i^0, \mathbf{y}_i^0, \dots, \mathbf{x}_{K-1}^0, \mathbf{y}_{K-1}^0, \dots, \mathbf{x}_0^j, \mathbf{y}_0^j, \dots, \mathbf{x}_{K-1}^j, \mathbf{y}_{K-1}^j, \dots, \mathbf{x}_0^{N-1}, \mathbf{y}_0^{N-1}, \dots, \mathbf{x}_{K-1}^{N-1}, \mathbf{y}_{K-1}^{N-1})}_{\delta^\top}$.

Avec cette convention, on a encore un bon aperçu de la forme de B :

Exemple dans le cas $K = 4$: La matrice B est de la forme :

$$B = \text{diag}(B_1, \dots, B_j, \dots, B_N) \text{ où } B_j = \begin{pmatrix} l & \mathbf{x}_0^j & \mathbf{y}_0^j & \mathbf{x}_1^j & \mathbf{y}_1^j & \mathbf{x}_2^j & \mathbf{y}_2^j & \mathbf{x}_3^j & \mathbf{y}_3^j & (i_1, i_2) \\ 6j & & & & & 0 & 0 & 0 & 0 & (1, 2) \\ 6j + 1 & & 0 & 0 & & & 0 & 0 & & (1, 3) \\ 6j + 2 & & & 0 & 0 & 0 & 0 & & & (1, 4) \\ 6j + 3 & 0 & 0 & & & & & 0 & 0 & (2, 3) \\ 6j + 4 & 0 & 0 & & & 0 & 0 & & & (2, 4) \\ 6j + 5 & 0 & 0 & 0 & 0 & & & & & (3, 4) \end{pmatrix}$$

Dans le cas général, B est donc une matrice diagonale par blocs avec $B_j \in \mathcal{M}_{\binom{K}{2}, 2N}$.

De même, les coefficients non nuls de B se calculent en utilisant les formules de la section précédente.

Remarque Par la suite, et comme nous allons le voir dans nos simulations, nous aurons besoin d'étudier notamment les conditionnements des matrices A et B .

4 Simulations

Avant d'appliquer notre méthode sur des images satellites réelles, nous devons la tester avec des simulations. Nous avons utilisé pour cela Python. On trouvera en annexe les codes relatifs à cette partie.

4.1 Le cas satellitaire

4.1.1 Génération de données

Dans la réalité, un satellite situé à environ 800 km de la Terre capture des images terrestres d'environ 20 km par 20 km. Afin de simuler au mieux des données réelles, on souhaite donc modéliser des scènes vérifiant les critères suivants :

- on veut que les caméras visent plus ou moins un même point au centre de la zone à modéliser, que l'on appellera point de visée,
- les points de correspondance identifiés par les algorithmes tels que la méthode SIFT doivent être situés dans la zone de 20 km par 20 km. Cependant, il ne faut pas négliger le relief terrestre : les points ne sont pas sur un plan, on laisse donc leur altitude varier de quelques kilomètres.

On génère ainsi aléatoirement N points de correspondance et K caméras correspondants à une telle situation.

L'étape la plus compliquée consiste à choisir les angles associés aux caméras pour qu'elles pointent le point de visée, que l'on définit arbitrairement comme le point $O = (0, 0, 0)$ (quitte à tout translater). Il est alors facile de voir (en utilisant des formules de changement de base) que, pour toute caméra i , le vecteur normalisé $\frac{O\vec{C}_i}{\|O\vec{C}_i\|}$ doit être l'image de $(0, 0, 1)^\top$ par la matrice de rotation R_i^{-1} associée à la caméra. De plus, on remarque que l'on peut fixer l'angle γ à 0 sans pour autant empêcher que certains vecteurs soient l'image de $(0, 0, 1)^\top$ par R_i^{-1} . Avec ce choix, on détermine alors $\alpha \in [0, 2\pi[$ et $\gamma \in [-\pi/2, \pi/2[$ satisfaisants (cf. notre code).

On obtient avec notre code des situations comme celle-ci :

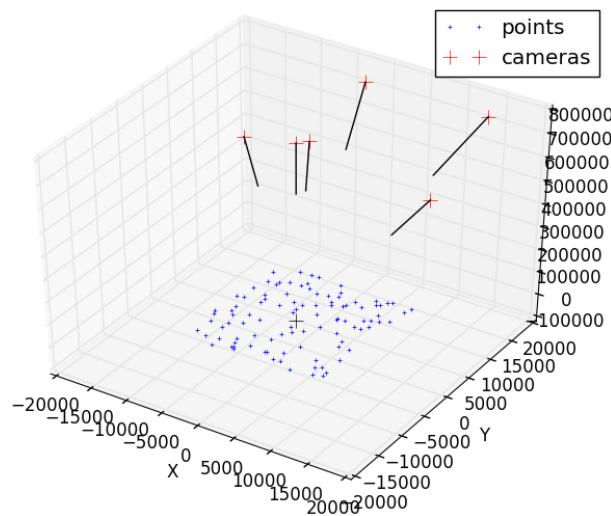


Figure 2: Une situation avec 6 caméras et 100 points
(les traits noirs sont les directions de visée des caméras)

Remarque Évidemment l'échelle de variation des points de correspondance est très faible par rapport à l'altitude des satellites. Les points semblent donc être sur un même plan. C'est une spécificité du cas satellitaire sur laquelle nous reviendrons plus tard.

4.1.2 Perturbation des données

A partir des points et des caméras créés, on peut, en utilisant les matrices de projection des caméras, obtenir les coordonnées réelles des images des points sur nos caméras. On connaît donc les valeurs réelles des angles θ_* et des images \mathbf{X}_* (cf. partie 1.3).

On perturbe alors ces points et ces angles en ajoutant un bruit gaussien de paramètres σ_x et σ_θ . Typiquement, on a $\sigma_x \simeq 10^{-2}$ px et $\sigma_\theta \simeq 10^{-5}$ rad.

On obtient alors θ_0 et \mathbf{X}_0 .

Remarque On rappelle que dans la réalité, on n'a pas accès à θ_* et \mathbf{X}_* , on nous fournit directement θ_0 et \mathbf{X}_0 . Ainsi, dans la suite du code, on ne pourra bien évidemment pas utiliser θ_* et \mathbf{X}_* pour nos calculs. On ne s'en servira que pour vérifier si nos estimations finales θ_a et \mathbf{X}_a sont plus proches ou non des données réelles que ne le sont θ_0 et \mathbf{X}_0 .

4.1.3 Estimation des paramètres réels

Ensuite, on utilise les données perturbées θ_0 et \mathbf{X}_0 à notre disposition pour calculer les matrices A et B , et donc la matrice M .

Sa pseudo-inverse permet alors de calculer nos estimations θ_a et \mathbf{X}_a .

Reste à vérifier que l'on obtient des résultats cohérents, c'est-à-dire que θ_a et \mathbf{X}_a soient plus proches de θ_* et \mathbf{X}_* que θ_0 et \mathbf{X}_0 et que l'ensemble des déterminants de F soient diminués significativement.

4.1.4 Premiers résultats

Remarque Afin de calculer la solution au moindre carrés de notre problème, nous avons dans un premier temps calculé la matrice pseudo-inverse de notre problème avec la fonction `pinv` du package `numpy.linalg`, mais nous avons finalement constaté que l'exécution de l'algorithme était plus rapide et que les résultats étaient toujours meilleurs en utilisant les matrices creuses (package `scipy.sparse`) et en calculant directement la solution aux moindres carrés sans passer par la pseudos-inverse avec la fonction `lsqr` du package `scipy.sparse.linalg`.

Afin de se faire une première idée sur la qualité des résultats que l'on obtient, on fait varier les différents paramètres (nombre de caméras, nombre de points, σ_x , σ_θ , ...). Pour chaque ensemble de paramètres fixés, on réalise plusieurs expériences (une quinzaine ici) de simulation de données et de réestimations. Pour chacune d'entre elle, on affiche alors sur un graphe les erreurs initiales et les erreurs après correction des différents indicateurs (erreurs selon les angles, selon les points et selon les déterminants). Plus précisément (exemple pour l'erreur angulaire) :

- en bleu, on affiche l'erreur initiale ($\theta_0 - \theta_*$),
- en rouge, on affiche l'erreur après correction ($\theta_a - \theta_*$),
- pour chaque vecteur d'erreur, on affiche avec des carrés l'erreur moyenne et avec des traits les erreurs maximale et minimale du vecteur.

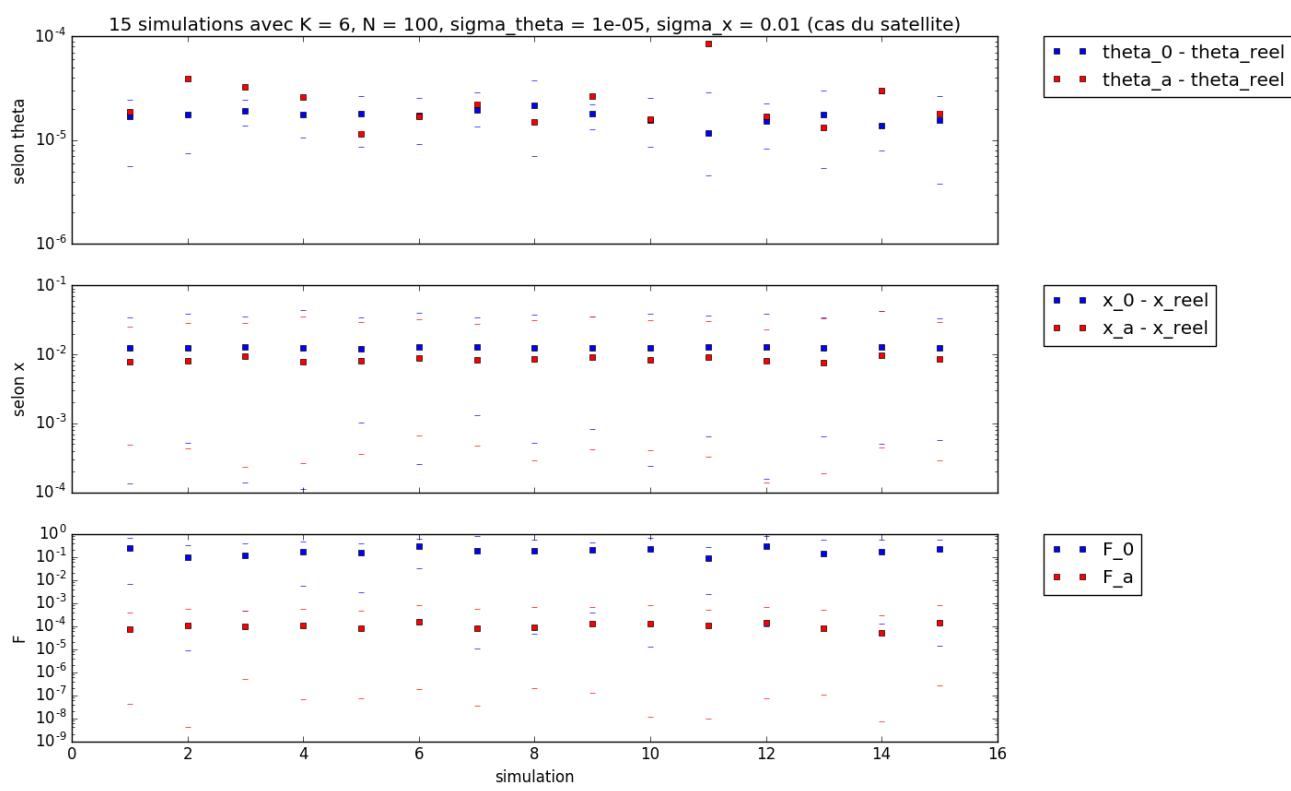


Figure 3: 15 simulations générales

Quelques soient les paramètres, on observe en général :

- que la valeur des déterminants est très bien diminuée par notre nouvelle estimation,
- que l'erreur des positions des images des caméras est légèrement diminuée, mais pas significativement,
- que l'erreur angulaire n'est pas souvent diminuée.

On remarque donc que l'on a un problème puisque l'on n'arrive pas (assez) à se rapprocher des paramètres réels. On arrive quand même à diminuer les valeurs de F , ce qui est mauvais signe puisque cela pourrait signifier que F prend des valeurs très faibles sur toute une famille de solutions proche de notre solution réelle, et donc qu'il est difficilement possible de retrouver la valeur attendue.

4.1.5 Étude plus approfondie : le conditionnement des matrices

Afin de déterminer la nature du problème, on le teste dans des situations moins compliquées : avec moins de caméras, moins de points ...

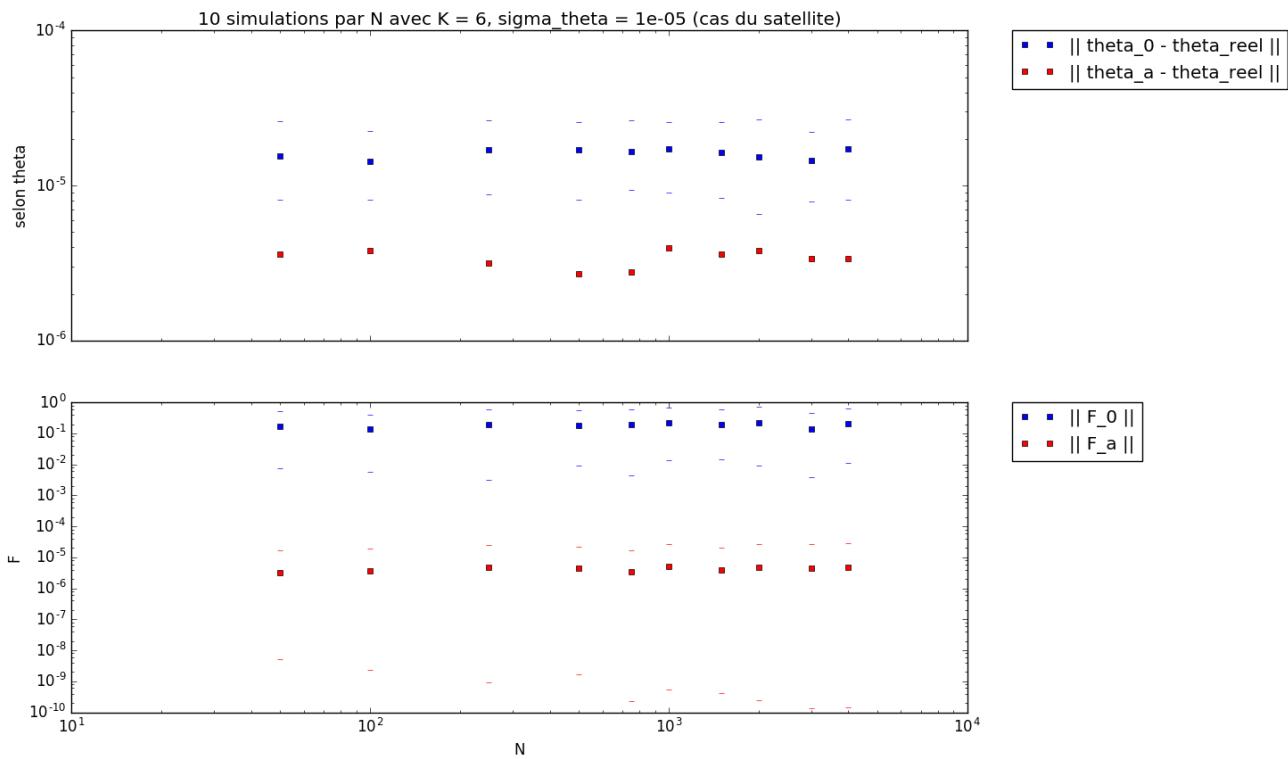
Une expérience importante consiste à voir ce qui se passe lorsque l'on essaye d'estimer uniquement les angles ou uniquement les coordonnées des images.

Tout d'abord, si l'**on fixe les coordonnées des images** et qu'**on essaye d'estimer uniquement les angles**, la matrice associée au problème (c'est-à-dire dont on doit calculer la pseudo-inverse) est la matrice A .

- Si l'on suppose que les coordonnées des images ne sont pas perturbées et donc que $\mathbf{X}_0 = \mathbf{X}_*$, on remarque que l'on arrive à diminuer les erreurs angulaires des caméras avec une bonne précision (on divise l'erreur par un facteur 3 ou 4 sur la figure ci-dessous). Les valeurs des déterminants sont elles aussi amplement diminuées par rapport aux erreurs initiales.

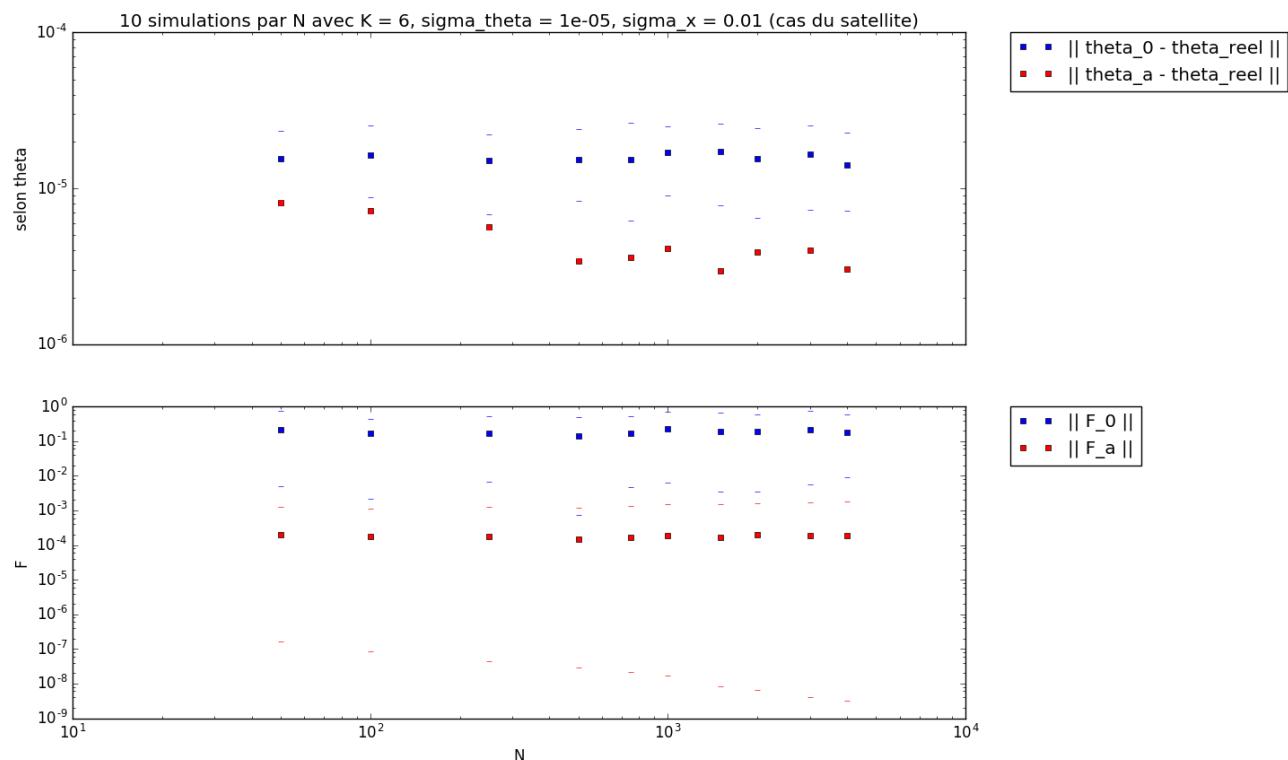
Sur le graphique suivant, on réalise, pour différentes valeurs de N , une dizaine de simulations (avec des coordonnées différentes à chaque fois) puis on calcule moyennes des erreurs des vecteurs $\theta_a - \theta_*$ puis

$F(\theta_a, \mathbf{X}_0 = \mathbf{X}_*)$ (en rouge ci-dessous) et $\theta_0 - \theta_*$ puis $F(\theta_0, \mathbf{X}_0 = \mathbf{X}_*)$ (en bleu).



On voit bien que notre estimation finale est beaucoup plus proche de la valeur réelle et que la norme de F a été fortement diminuée. Par ailleurs, le nombre de points N ne semble pas avoir un rôle particulier.

- On peut aussi supposer qu'on a eu une perturbation des coordonnées des images (ce qui est le cas dans la réalité) mais que l'on ne cherche pas à la corriger. Dans ce cas, on a donc $\mathbf{X}_0 \neq \mathbf{X}_*$. On obtient alors des résultats du type :

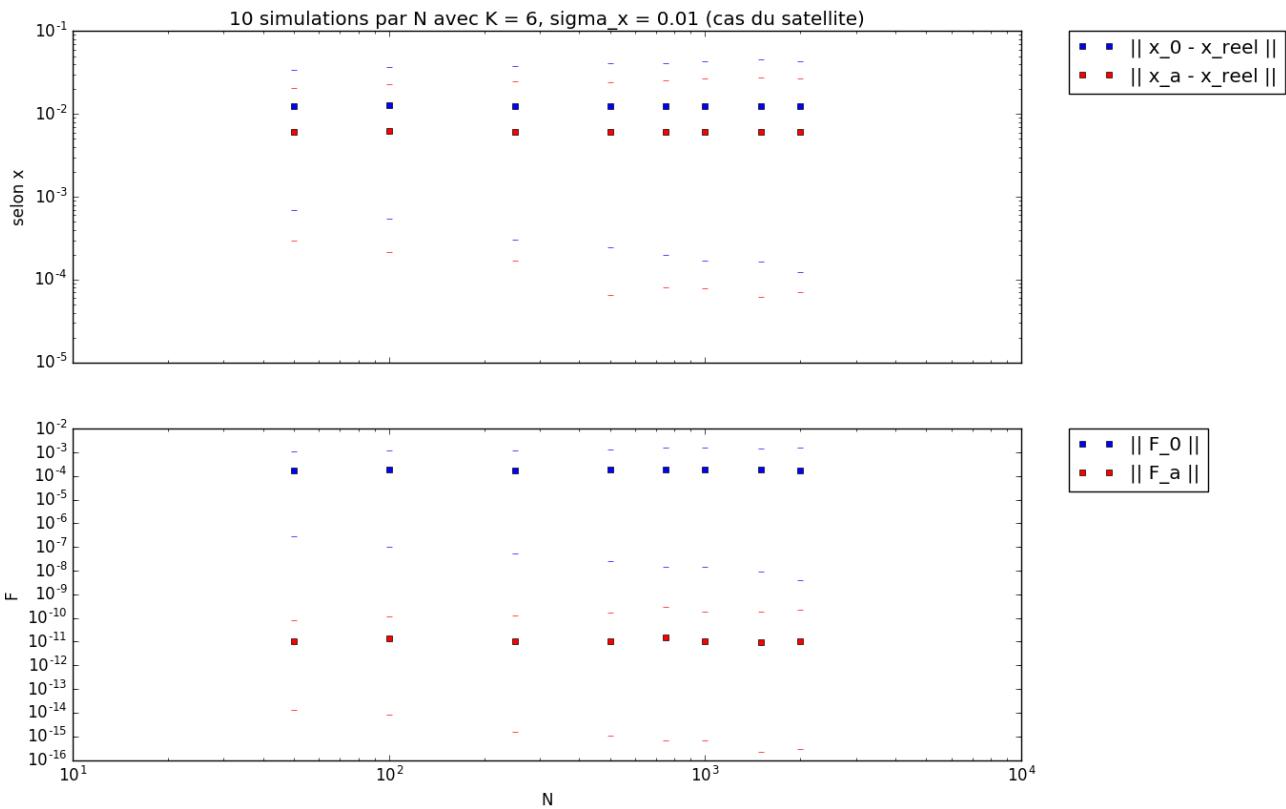


Cette fois-ci, on arrive à réestimer un peu mieux les angles, mais on observe surtout que la qualité de la réestimation dépend du nombre de points N : plus l'on a de points et plus on va pouvoir réestimer les angles précisément.

C'est à la fois une bonne nouvelle puisque l'on a en général accès à plusieurs centaines de points (voire quelques milliers) mais aussi une limitation puisque c'est les algorithmes de type SIFT qui fournissent les points de correspondance et qui limitent le nombre de points.

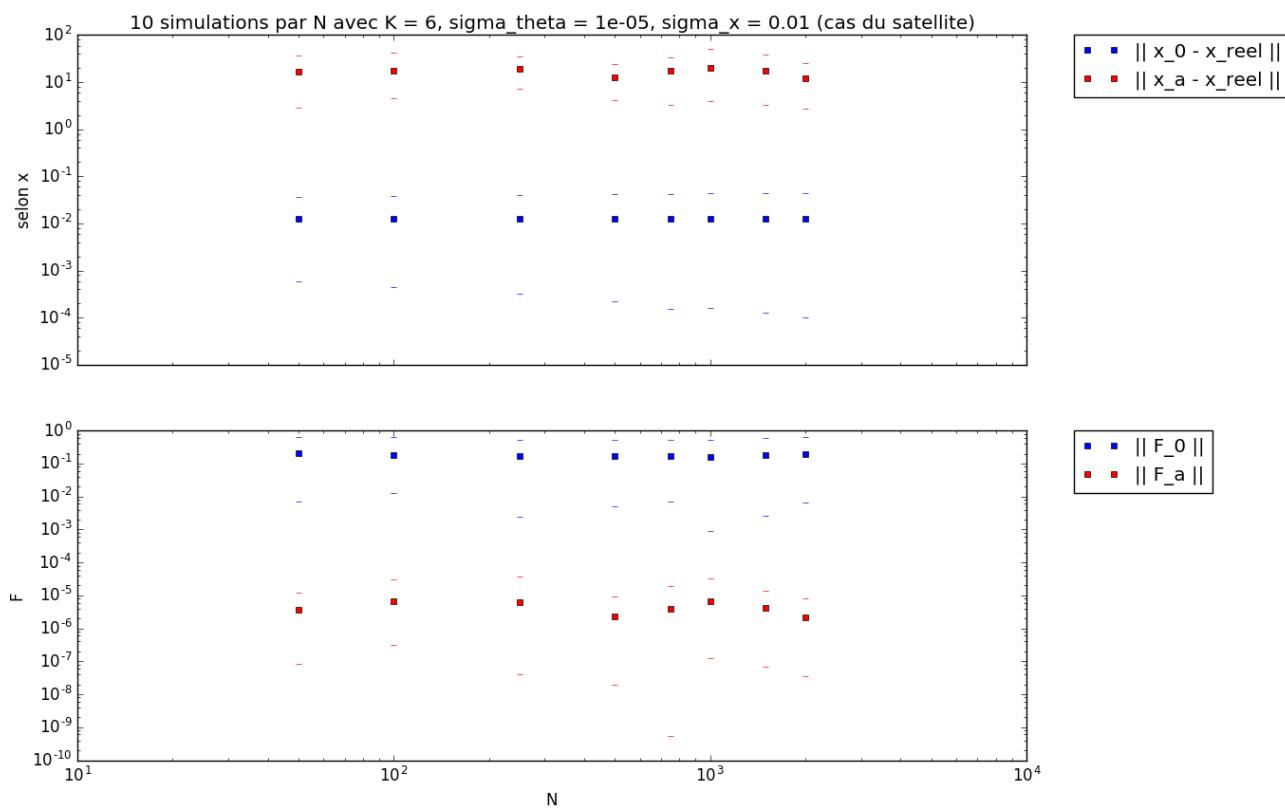
Ensuite, on peut faire l'expérience inverse : **on fixe les angles des caméras et on essaye d'estimer uniquement les coordonnées des images**. La matrice associée au problème est alors la matrice B .

- Si l'on suppose $\theta_0 = \theta_*$ (on n'a pas d'erreur sur nos angles), on obtient par exemple :



On observe que notre réestimation se rapproche légèrement de la solution initiale, sans pour autant que cela soit très significatif. La valeur des déterminants est quant à elle très fortement diminuée. Enfin, le nombre de points N ne semble pas avoir de rôle particulier.

- Si l'on rajoute une petite perturbation sur les angles, comme dans la réalité, on obtient :



Cette fois-ci, les résultats ne sont pas très bons : les coordonnées réestimées s'éloignent des coordonnées initiales ...

Comment interpréter ces résultats ?

→ On peut s'attendre à un **problème de conditionnement** des différentes matrices.

Regardons alors les conditionnements des matrices A , B et M , pour une dizaine de matrices aléatoires pour chaque valeur de N testée :

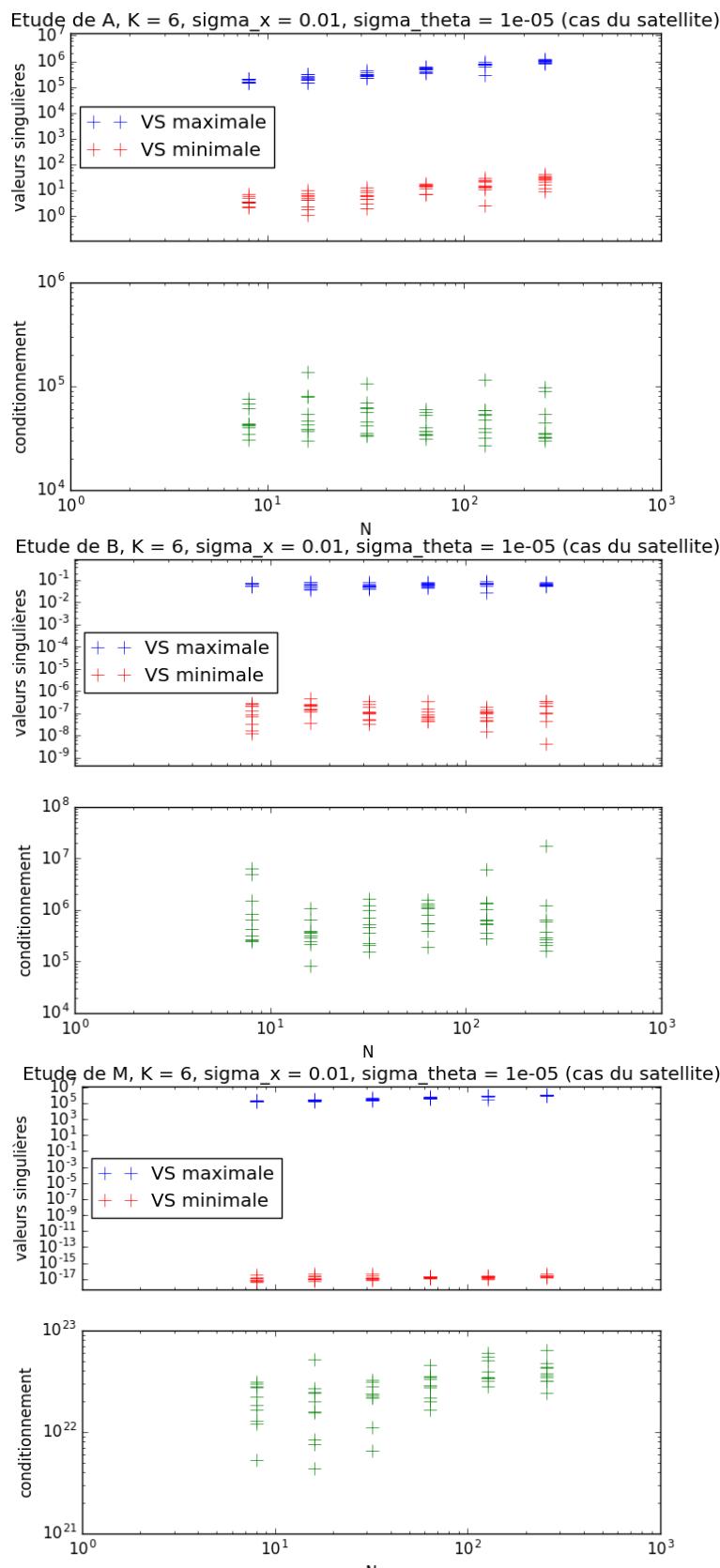


Figure 4: Étude des conditionnements des matrices $A^T A$, $B^T B$ et $M^T M$

On observe de très mauvais conditionnements pour le problème général associé à la matrice M (où l'on doit réestimer tous les paramètres). Les deux autres conditionnements pour les problèmes avec paramètres fixés sont moins monstrueux mais restent très importants.

4.2 Un cas terrestre

Essayons de voir ce que donne notre méthode si l'on change les ordres de grandeurs, en essayant par exemple de reconstituer une scène avec des images prises par des drones.

4.2.1 Génération de données

On procède un peu de la même manière pour générer nos données. Évidemment, on change tous les ordres de grandeurs : la scène s'étend sur une zone d'environ 100 m par 100 m, sur une hauteur d'une quarantaine de mètres environ.

On obtient avec notre code des situations comme celle-ci :

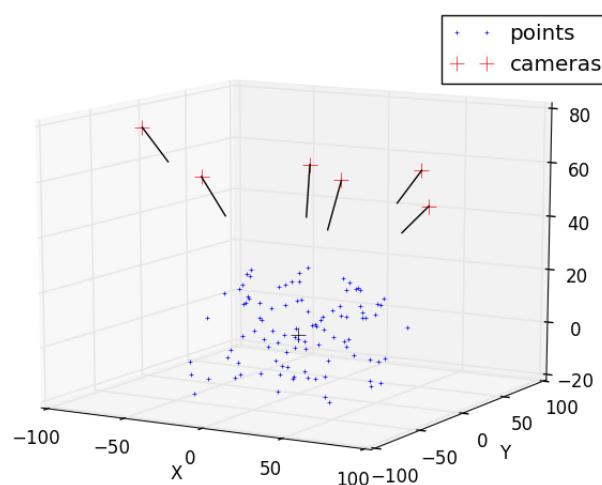


Figure 5: Une situation avec 6 caméras et 100 points
(les traits noirs sont les directions de visée des caméras)

Cette fois-ci, les points de correspondances se situent dans un pavé et ne semblent plus être sur un même plan. Cela pourrait-il laisser présager un meilleur conditionnement des matrices ?

4.2.2 Perturbation des données

On perturbe de même nos données en ajoutant un bruit gaussien de paramètres σ_x et σ_θ . Les ordres de grandeurs des perturbations sont ici différents, on prend typiquement $\sigma_x \simeq 10^{-2}$ px et $\sigma_\theta \simeq 10^{-2}$ rad.

4.2.3 Estimation des paramètres réels

Là encore, on utilise les données perturbées θ_0 et \mathbf{X}_0 à notre disposition pour calculer les matrices A et B , et donc la matrice M .

4.2.4 Premiers résultats

Quelques soient les paramètres, on observe en général :

- que la valeur des déterminants est bien diminuée par notre nouvelle estimation,

- que la position des images des caméras n'est pas du tout bien réestimée, puisque l'erreur augmente d'un facteur environ 10^3 après notre estimation,
- que l'erreur angulaire reste au moins du même ordre qu'à l'origine, voire dans certains cas est bien améliorée.

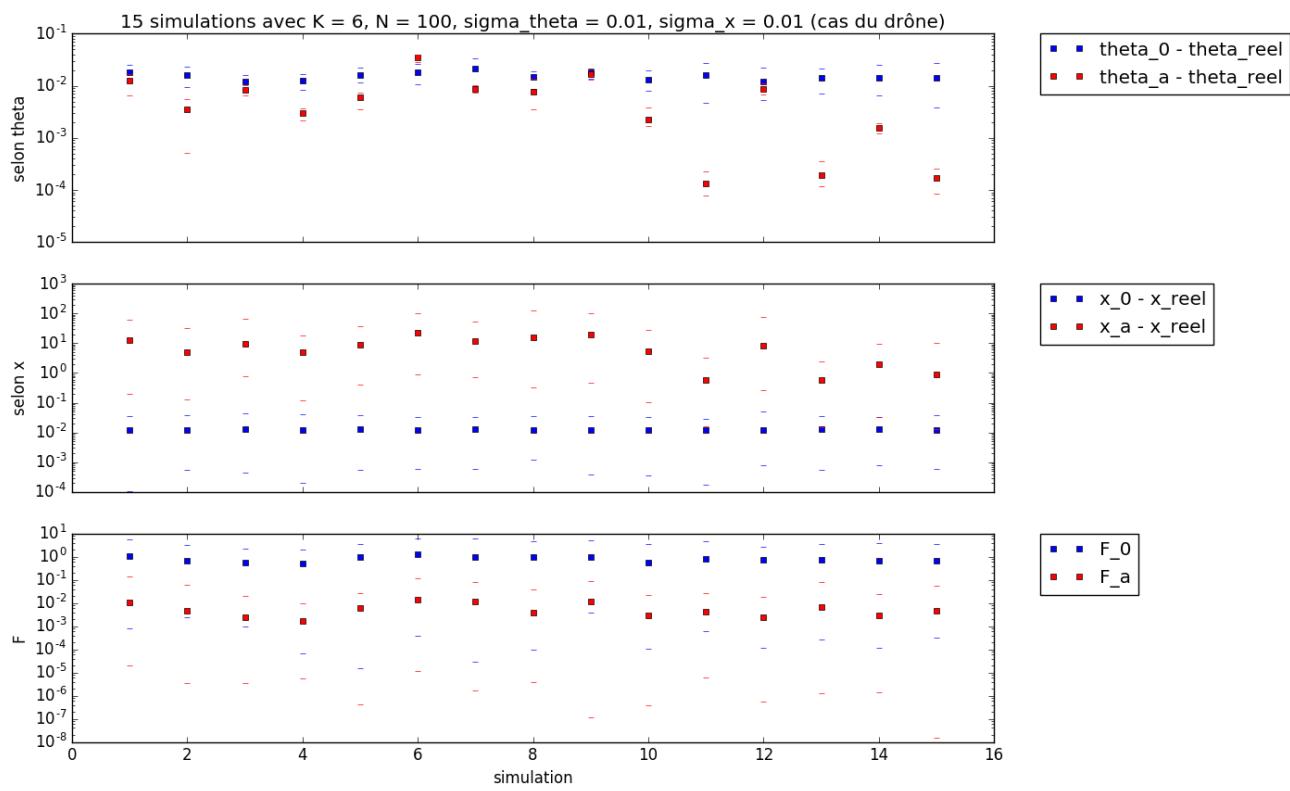


Figure 6: 15 simulations générales

4.2.5 Étude plus approfondie : le conditionnement des matrices

Tout d'abord, si l'on fixe les coordonnées des images et qu'on essaye d'estimer uniquement les angles :

- Si l'on suppose que les coordonnées ne sont pas perturbées et donc que $\mathbf{X}_0 = \mathbf{X}_*$, on remarque que l'on arrive toujours à retrouver les angles réels des caméras avec une très grande précision.
On voit que notre estimation finale est beaucoup plus proche de la valeur réelle et que la norme de F a été fortement diminuée.
- On peut aussi supposer qu'on a eu une perturbation des coordonnées des images mais qu'on ne cherche pas à la corriger. Dans ce cas, on a donc $\mathbf{X}_0 \neq \mathbf{X}_*$. On obtient alors des résultats très similaires :
On arrive donc une nouvelle fois à retrouver les paramètres angulaires réels avec une bonne précision. **C'est mieux que dans le cas satellitaire, puisque cette expérience ne fonctionnait qu'à partir d'un nombre important de points !**

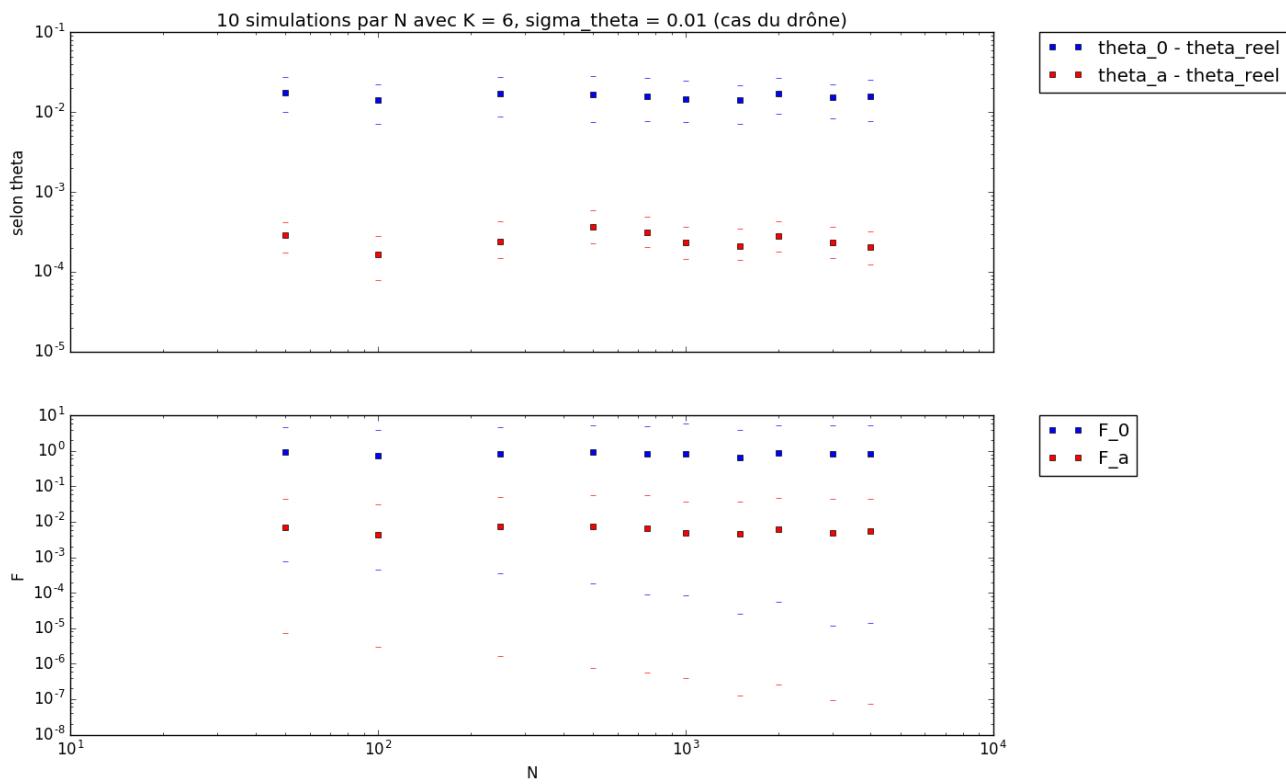


Figure 7: Évolution des différentes normes associées au problème en fonction du nombre de points

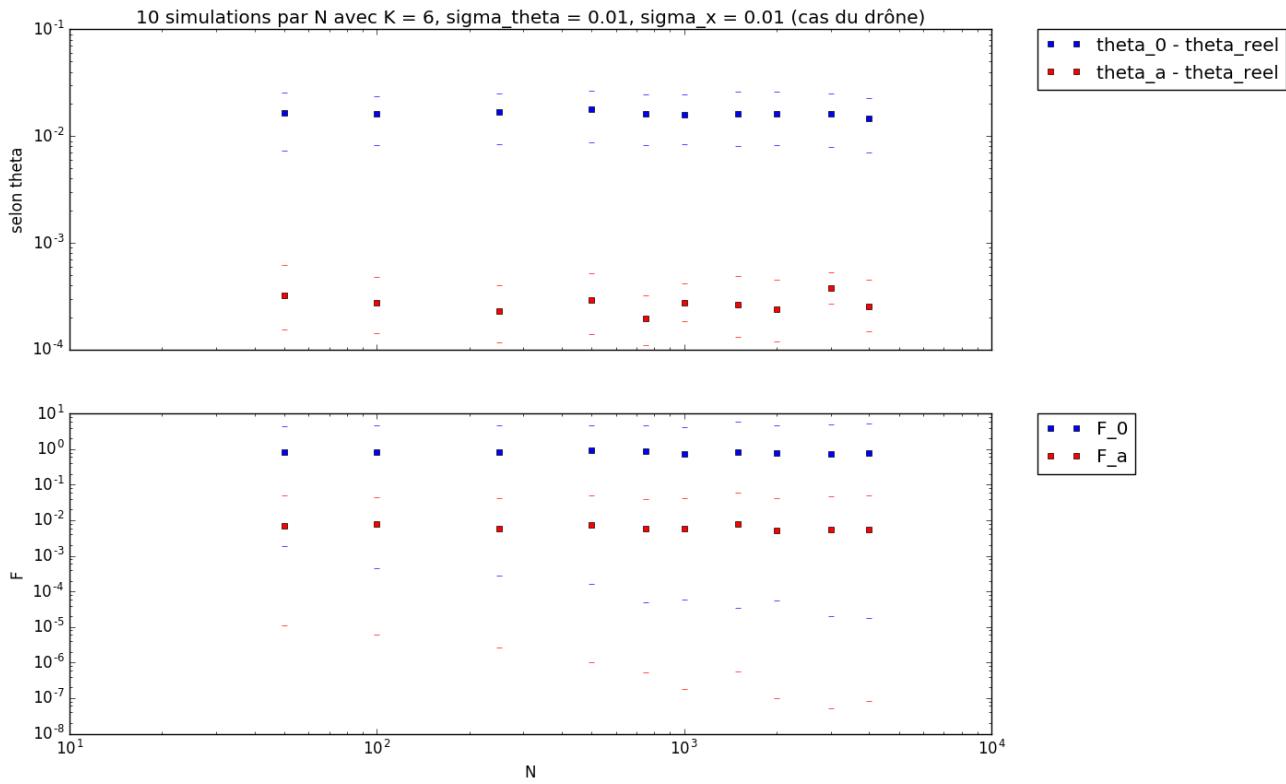


Figure 8: Évolution des différentes normes associées au problème en fonction du nombre de points

Ensuite, on peut faire l'expérience inverse : on fixe les angles des caméras et on essaye d'estimer uniquement les coordonnées des images. La matrice associée au problème est alors la matrice B .

Si l'on suppose $\theta_0 = \theta_*$ (on n'a pas d'erreur sur nos angles), on obtient par exemple :

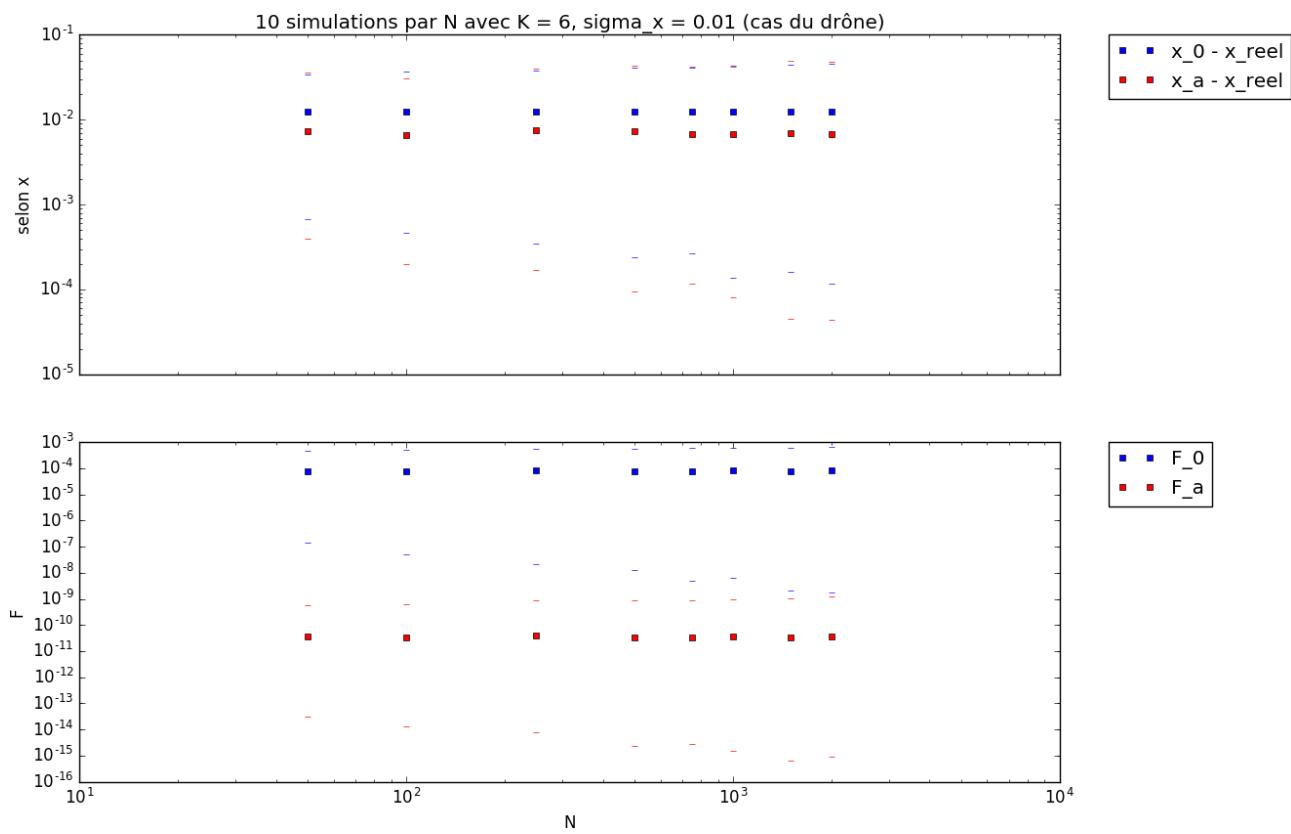


Figure 9: Évolution des différentes normes associées au problème en fonction du nombre de points

Comme dans le cas satellitaire, on arrive bien à minimiser F mais on n'arrive pas à se rapprocher des coordonnées réelles.

Regardons alors les conditionnements des matrices A , B et M , pour une dizaine de matrices aléatoires pour chaque valeur de N testée :

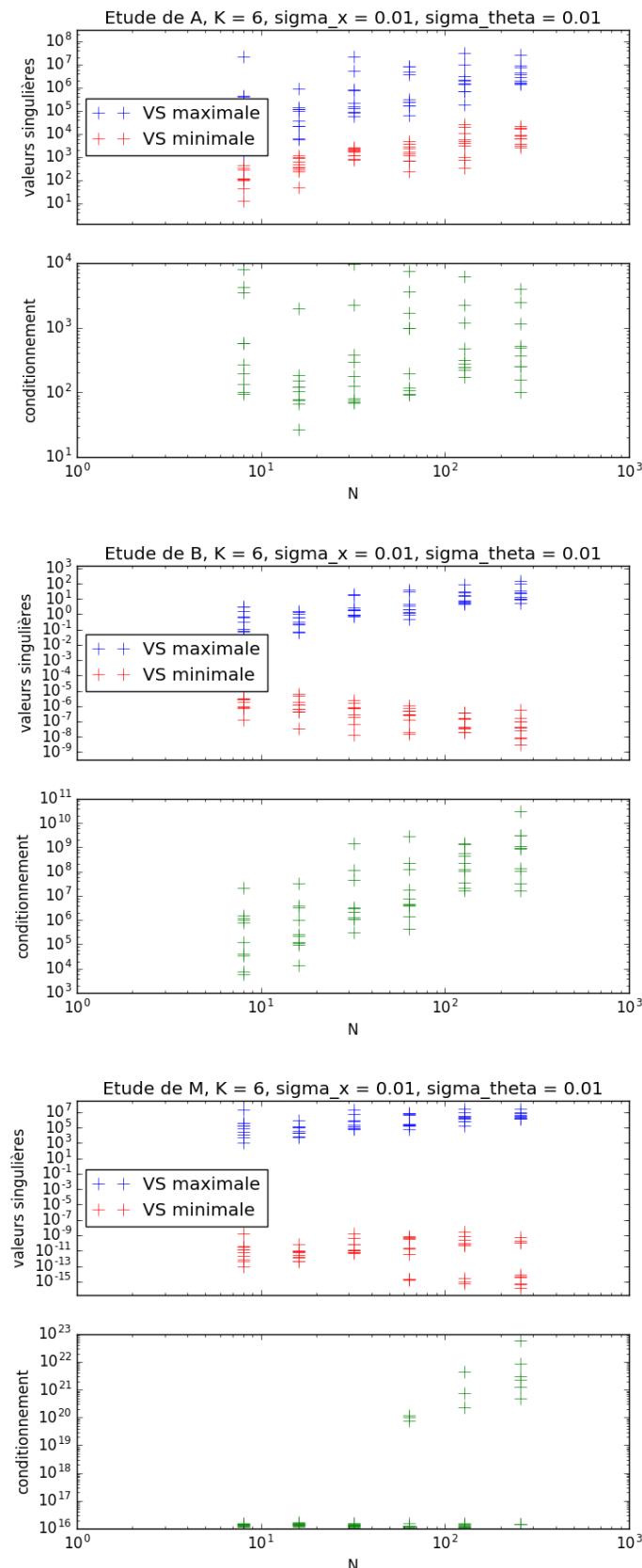


Figure 10: Étude des conditionnements des matrices $A^T A$, $B^T B$ et $M^T M$

On observe encore de très mauvais conditionnements pour le problème général associé à la matrice M . Le conditionnement de la matrice B (estimation des coordonnées des images) est lui aussi assez mauvais. Pour la

matrice A , le conditionnement est assez aléatoire mais pas trop important.

4.3 Améliorations de la méthode : ajout d'un terme forçant

4.3.1 Principe

Les expériences précédentes montrent que notre problème est mal conditionné, et que plusieurs approximations sont possibles au voisinage des données initiales. On espérait que la résolution, au sens des moindres carrés, du système

$$M \begin{pmatrix} d\theta \\ d\mathbf{X} \end{pmatrix} = -F(\theta_0, \mathbf{X}_0)$$

nous conduirait à l'obtention d'une unique solution, à savoir la solution (θ_*, \mathbf{X}_*) mais ce n'est clairement pas le cas puisque le simulateur nous renvoie des valeurs (θ_a, \mathbf{X}_a) qui diminuent les valeurs des déterminants mais qui restent loin de la solution réelle et même parfois de la solution initiale.

Afin de corriger ce problème, on impose une contrainte supplémentaire au problème en cherchant des solutions $(d\theta, d\mathbf{X})$ qui minimisent la quantité :

$$\left\| M \begin{pmatrix} d\theta \\ d\mathbf{X} \end{pmatrix} + F(\theta_0, \mathbf{X}_0) \right\|^2 + \frac{\lambda^2}{\sigma^2} \left\| \begin{pmatrix} d\theta \\ d\mathbf{X} \end{pmatrix} \right\|^2 \quad (2)$$

où λ est un préfacteur à déterminer afin d'obtenir des résultats optimaux.

Cette nouvelle contrainte permet de mieux contrôler la distance entre (θ_0, \mathbf{X}_0) et (θ_a, \mathbf{X}_a) . On espère donc obtenir une approximation finale meilleure que sur nos premières simulations.

Remarque En pratique, on n'a pas $\sigma_{\mathbf{X}} = \sigma_\theta$, et on n'est pas obligé d'imposer $\lambda_{\mathbf{X}} = \lambda_\theta$. On essayera donc plutôt de minimiser la quantité :

$$\left\| M \begin{pmatrix} d\theta \\ d\mathbf{X} \end{pmatrix} + F(\theta_0, \mathbf{X}_0) \right\|^2 + \frac{\lambda_\theta^2}{\sigma_\theta^2} \|d\theta\|^2 + \frac{\lambda_{\mathbf{X}}^2}{\sigma_{\mathbf{X}}^2} \|d\mathbf{X}\|^2$$

4.3.2 Implémentation

Pour mettre en place cette méthode, on rajoute à notre système linéaire les équations :

$$\frac{\lambda_\theta}{\sigma_\theta} d\theta = 0 \text{ et } \frac{\lambda_{\mathbf{X}}}{\sigma_{\mathbf{X}}} d\mathbf{X} = 0$$

Cela revient à rajouter le bloc $\text{diag}(\sigma_\theta, \dots, \sigma_\theta, \sigma_{\mathbf{X}}, \dots, \sigma_{\mathbf{X}}) \in \mathcal{M}_{3K+2KN}$ à la matrice M . On obtient ainsi une nouvelle matrice M' qui possède plus de lignes (plus d'équations) et on résout le système par la méthode habituelle, la pseudo-inverse de M' permettant alors de minimiser la quantité (2).

Remarque Algorithmiquement, cela rallonge considérablement le temps de calcul.

4.3.3 Résultats

Dans la suite, on utilisera les indices suivants :

- 0 sera associé aux données initialement perturbées,

- "réel" sera associé aux données exactes,
- a_1 sera associé à la solution fournie par l'algorithme sans terme forçant,
- a_2 sera associé à la solution fournie par l'algorithme avec terme forçant.

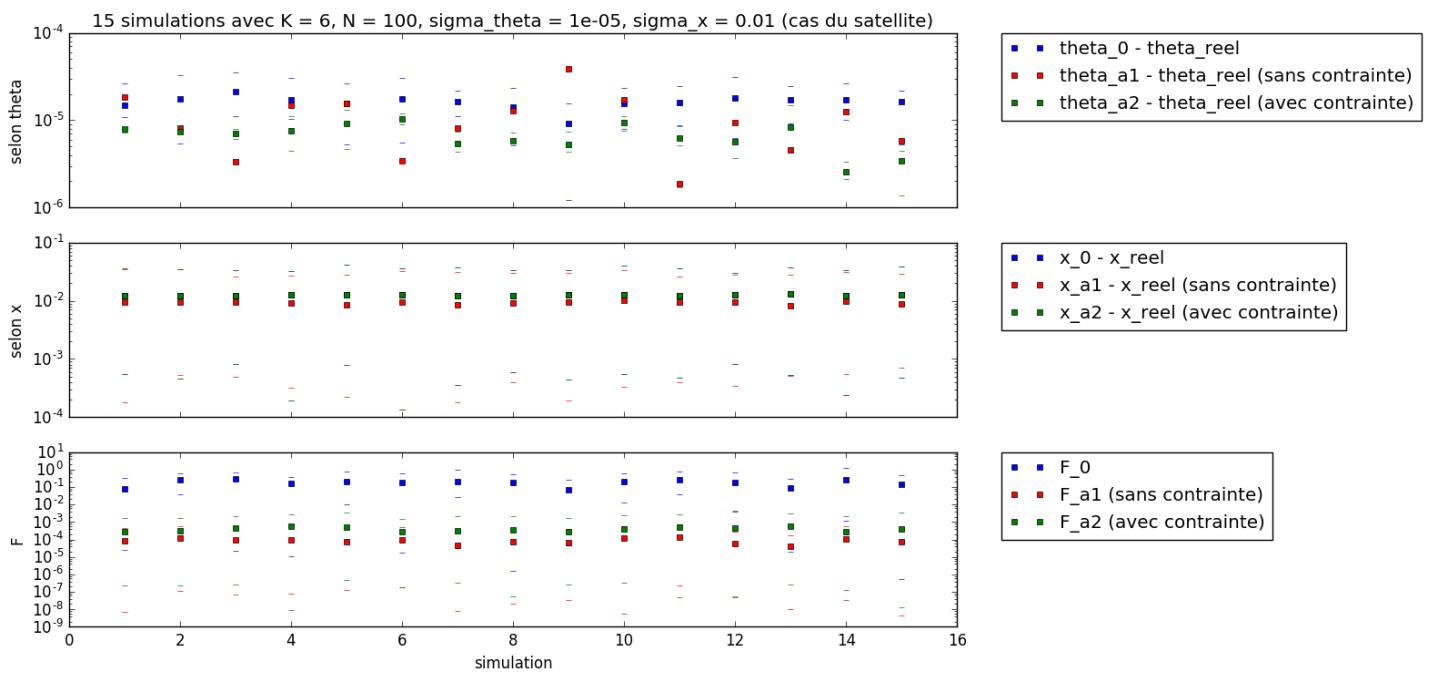


Figure 11: Comparaison sur F , x et θ des solutions a_1 et a_2

On remarque que l'algorithme sans contrainte minimise un peu mieux la norme de F que l'algorithme avec contrainte, ce qui s'explique simplement par le fait que l'algorithme sans contrainte est autorisé à aller chercher des solutions plus loin de la solution initiale. Cependant pour les deux algorithmes la minimisation est satisfaisante (les déterminants sont environ divisés par 1000).

Selon x , la solution avec terme forçant est identique à la solution initiale et légèrement moins bonne que la solution sans terme forçant. En revanche le terme forçant contribue nettement à rapprocher la solution des angles exactes alors qu'en l'absence de terme forçant aucun comportement régulier n'est observé.

Remarque Pour se rapprocher encore davantage de la solution réelle selon les angles, on peut alors essayer de procéder à plusieurs itérations de l'algorithme, mais on remarque que cela n'améliore pas beaucoup plus la solution.

Ainsi nos observations nous assurent que notre méthode d'approximation des angles des caméras et des coordonnées des points fonctionne : notre algorithme nous fournit une solution qui est plus proche de la solution réelle que la solution initiale.

4.3.4 Quelques tests complémentaires

Voici quelques résultats obtenus en faisant varier :

- les conditions initiales (on fixe certaines valeurs initiales à leur valeur réelle)
- les paramètres que l'on réestime (c'est-à-dire ceux qu'on obtient après inversion de la matrice M)

On a fixé $N = 100$, $K = 6$, $\sigma_x = 0.1$, $\sigma_\theta = 0.00001$, $\lambda_x = \lambda_\theta = 0.02$. On notera $dist_x$ la distance de la solution à la solution réelle (norme de Frobenius) selon x , et $dist_\theta$ a distance de la solution à la solution réelle selon θ .

Les tests ont été réalisés avant et après rajout du terme forçant.

Paramètres fixés à leur valeur réelle	Paramètres non réestimés	Résultats sans contrainte	Résultats avec contrainte
Aucun	Aucun	<ul style="list-style-type: none"> - F: bien diminué (facteur 10^2) - $dist_x$ diminué (facteur 2 qui ne varie pas avec N) - $dist_\theta$ augmenté, tend vers la solution initiale lorsque N augmente 	<ul style="list-style-type: none"> - F: bien diminué (facteur 10^3) - $dist_x$ diminué (facteur 2 qui ne varie pas avec N) - $dist_\theta$ diminution (facteur 2 qui ne varie pas avec N)
x	Aucun	<ul style="list-style-type: none"> - F: bien diminué (facteur 10^2) - $dist_x$: identique à la solution initiale - $dist_\theta$: pas d'amélioration 	<ul style="list-style-type: none"> - F: bien diminué (facteur 10^4) - $dist_x$: quasi identique à la solution initiale - $dist_\theta$: diminué (facteur 2)
x	x	<ul style="list-style-type: none"> - F: bien diminué (facteur 10^4) - $dist_x$: identique à la solution initiale - $dist_\theta$: nette diminution (facteur 10^2) 	<ul style="list-style-type: none"> - F: bien diminué (facteur 10^3) - $dist_x$: identique à la solution initiale - $dist_\theta$: diminué (facteur 2)
Tous les x sauf ceux sur camera 1	Tous les x sauf ceux sur camera 1	<ul style="list-style-type: none"> - F: bien diminué (facteur 10^3) - $dist_x$: amélioré (facteur 10) - $dist_\theta$: pas d'amélioration régulière 	<ul style="list-style-type: none"> - F: bien diminué (facteur 10^3) - $dist_x$: amélioration (facteur 10) - $dist_\theta$: amélioration (facteur 2)
Tous les x sauf ceux sur camera 1	Tous les x sauf ceux sur camera 1 et θ	<ul style="list-style-type: none"> - F: presque pas diminué - $dist_x$: augmenté (facteur 10). - $dist_\theta$: identique solution initiale 	<ul style="list-style-type: none"> - F: diminué (facteur 100) - $dist_x$: diminué (facteur 10). - $dist_\theta$: identique solution initiale
Tous les x sauf ceux sur camera 1 et θ	Tous les x sauf ceux sur camera 1 et θ	<ul style="list-style-type: none"> - F: beaucoup diminué (facteur 10^8) - $dist_x$: beaucoup diminué (facteur 10^8). - $dist_\theta$: identique solution initiale 	<ul style="list-style-type: none"> - F: diminué (facteur 10^2) - $dist_x$: diminué (facteur 10^2). - $dist_\theta$: identique solution initiale
Tous les x sauf ceux sur camera 1 et θ sauf les angles de la caméra 1	Tous les x sauf ceux sur camera 1 et θ sauf les angles de la caméra 1	<ul style="list-style-type: none"> - F: diminué (facteur 10^2) - $dist_x$: diminué (facteur 10) - $dist_\theta$: diminué (facteur 10) 	<ul style="list-style-type: none"> - F: diminué (facteur 10^2) - $dist_x$: diminué (facteur 10) - $dist_\theta$: diminué (facteur 10)
θ	θ	<ul style="list-style-type: none"> - F: diminué (facteur 10^6) - $dist_x$: pas d'amélioration - $dist_\theta$: identique à la solution initiale 	<ul style="list-style-type: none"> - F: diminué (facteur 10^2) - $dist_x$: diminué (facteur 2) - $dist_\theta$: identique à la solution initiale
θ et tous les x sauf ceux qui correspondent aux projections du point X_1 de l'espace	θ et tous les x sauf ceux qui correspondent aux projections du point X_1 de l'espace	<ul style="list-style-type: none"> - F: diminué (facteur 10^6) - $dist_x$: pas d'amélioration - $dist_\theta$: identique à la solution initiale 	<ul style="list-style-type: none"> - F: diminué (facteur 10^2) - $dist_x$: diminué (facteur 2) - $dist_\theta$: identique à la solution initiale

5 Application à des images satellitaires

Afin de mieux comprendre l'intérêt de notre méthode et de nos hypothèses par rapport aux techniques classiques de bundle adjustment, nous nous sommes intéressés à des images satellitaires réelles, prises sur la ville de Napier en Nouvelle-Zélande. Nous avons voulu comparer les résultats de notre méthode aux résultats obtenus avec les méthodes de Bundle Adjustment classiques.

Nous avons à notre disposition 20 images du satellite Pléiades prises lors à différentes dates (on a des doublets et des triplets d'images). On nous fournit également une approximation des 20 matrices de projection.

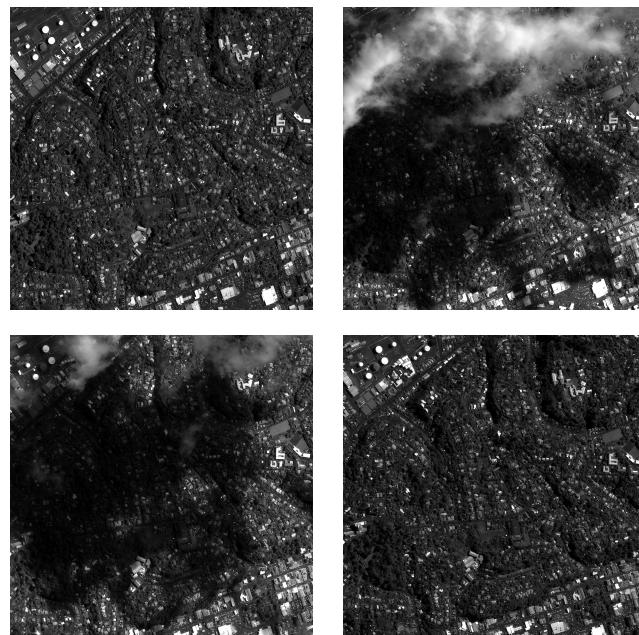


Figure 12: Quelques images de la ville de Napier

5.1 Implémentation de notre méthode

5.1.1 Obtention de correspondances à 6 points à partir d'un jeu de 6 images

La première étape consiste à sélectionner 6 images parmi les 20 et à en extraire des points de correspondances. Alors qu'un algorithme SIFT nous fournit des correspondances deux à deux entre les images, notre algorithme nécessite de connaître des correspondances entre nos 6 images. Pour ce faire nous avons créé des correspondances à 6 images à partir des correspondances deux à deux en utilisant le fait que si un point x_1 dans l'image 1 correspond au point x_2 dans l'image 2, et que x_2 correspond au point x_3 dans l'image 3, alors x_1 et x_3 sont en correspondance.

Chercher l'ensemble des correspondances à 6 points revient à déterminer les composantes connexes à 6 éléments du graphe tel que:

- ses sommets sont de la forme (x, y, k) où (x, y) est la coordonnée d'un point dans l'image k .
- une arête entre deux sommets signifie que les deux points (représentés par les deux sommets) sont en correspondance.

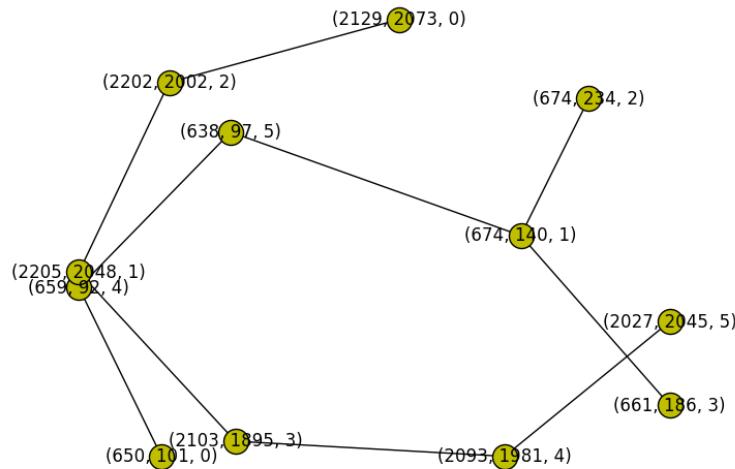


Figure 13: Exemple de graphe (réduit) correspondant au problème

L'implémentation d'un algorithme pour déterminer des correspondances six à six devient alors plus simple en utilisant un package Python spécifique aux graphes.

5.1.2 Sélection des meilleurs jeux d'images pour l'implémentation

Plus l'on a de points de correspondance, et plus l'on s'attend (cf. nos simulations) à gagner en précision. Le choix des caméras utilisées pour réestimer la scène 3D est donc très important puisque les longueurs des fichiers de correspondance deux à deux diffèrent beaucoup !

On souhaite donc déterminer les jeux de K images parmi les 20 pour lesquels on aura le plus de points de correspondances. Pour cela, on regarde le nombre de correspondances deux à deux pour toutes les caméras :

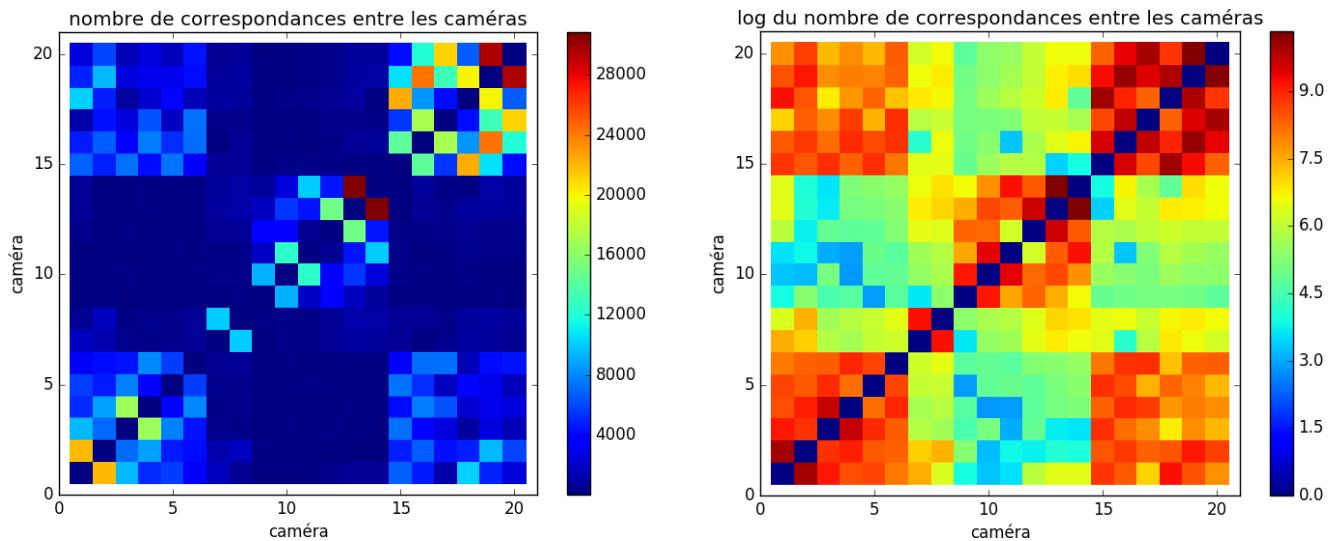


Figure 14: Matrice des correspondances (ransac 0,2)

On a trois types de fichiers de correspondance, selon le filtrage Ransac effectué (aucun, 0,2 px ou 0,05 px). Les matrices ci-dessus sont globalement peu modifiées par le type de filtrage (à un rapport près sur le nombre de correspondances).

On pourrait implémenter un algorithme qui estime les meilleures caméras à utiliser. Cependant, dans notre cas ($K = 6$ la plupart du temps), on peut voir assez facilement à la main quels K -uplets de caméras semblent bien fonctionner.

Afin de montrer l'importance d'un bon choix des caméras, prenons quelques 6-uplets de caméras pour lesquels le nombre de correspondances peut varier significativement :

6-uplet	Sans filtrage	Ransac 0,2	Ransac 0,05
(02, 15, 16, 17, 18, 19)	10504	2311	42
(01, 02, 03, 04, 05, 06)	4751	1193	19
(12, 13, 14, 18, 19, 20)	1608	280	6
(01, 02, 07, 08, 13, 14)	1181	185	1
(07, 08, 09, 10, 11, 12)	876	88	2

Dans la suite de nos expériences, nous utiliserons sauf mention contraire les données filtrées à 0,2 px.

5.1.3 Méthodes pour évaluer les résultats

Dans le cas d'images réelles, nous n'avons accès ni aux angles exacts ni aux coordonnées exactes. Il devient alors plus difficile d'évaluer la qualité de notre approximation, celle-ci étant considérée comme bonne si elle est proche des données exactes.

Comparaison de la solution à partir de différents ensembles de points. Les images de Napier nous fournissent environ 1200 correspondances (c'est-à-dire 1200 points de l'espace) pour les caméras 1, 2, 3, 4, 5, 6. Fixons une correspondance de référence que l'on appellera x_{ref} : x_{ref} est l'ensemble des projections d'un point x_{ref} de l'espace sur les 6 caméras. Prenons M ensembles de 200 correspondances ayant pour intersection deux à deux x_{ref} , et appliquons notre algorithme à ces M ensembles. Alors plus les M approximations de x_{ref} sont proches, plus la solution selon les x pourra être considérée comme bonne.

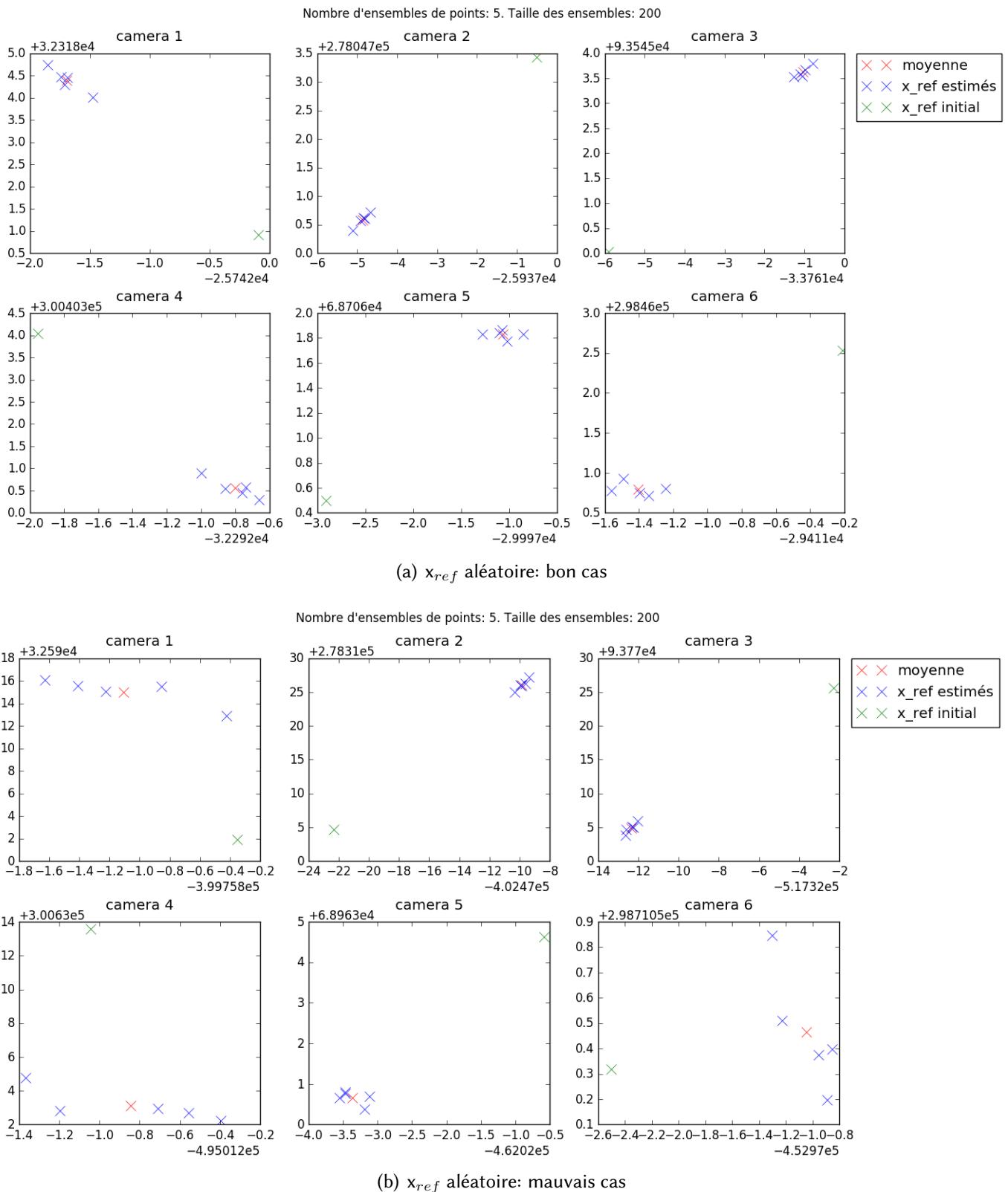
Pour nos tests nous avons pris $M = 5$. On a représenté x_{ref} (valeur initiale), les $x_{ref,a}$ estimés par les 5 ensembles de points et leur moyenne dans chacune des images.

On remarque qu'il existe des ensembles de points pour lesquels on obtient de très bons résultats (voir Figure (a)) c'est-à-dire la distance maximale entre les approximations est très inférieure à la distance moyenne à la solution initiale. Cependant pour certains rares x_{ref} (Figure (b)) on obtient des approximations trop dispersées pour pouvoir en déduire l'exactitude de nos solutions.

De la même façon, toujours avec les mêmes ensembles de points, on peut évaluer la qualité de nos approximations selon les angles. Nous avons représenté sur des droites pour chaque caméra et chaque direction X, Y, Z , les valeurs des estimations et la valeur initiale.

Encore une fois les résultats sont positifs: toutes nos solutions estimées à partir des différents ensembles de points sont groupées et à distance de la solution initiale.

Comparaison de la solution à partir de différents ensembles de cameras. Pour avoir une idée de la qualité de notre approximation selon les angles, on peut reprendre le même raisonnement que ci-dessus mais cette fois comparer les solutions obtenues à partir de différents ensembles des caméras se recouvrant en une caméra. Par exemple comparons les estimations des angles de la caméra 2 à partir des ensembles de caméras (1, 2, 3, 4, 5, 6) et (2, 15, 16, 17, 18, 19).

Figure 15: Distances des différentes solutions à la solution initiale selon x_{ref}

5.2 Comparaison avec les méthodes classiques de Bundle Adjustment

Un algorithme de Bundle Adjustment est un algorithme qui, étant donné des conditions initiales $\hat{\mathbf{x}}$ et \hat{P} (matrices de projection) minimise sur les matrices de projection $(P_i)_{0 \leq i \leq K}$ et sur les points de l'espace $(\mathbf{X}^j)_{0 \leq j \leq N}$

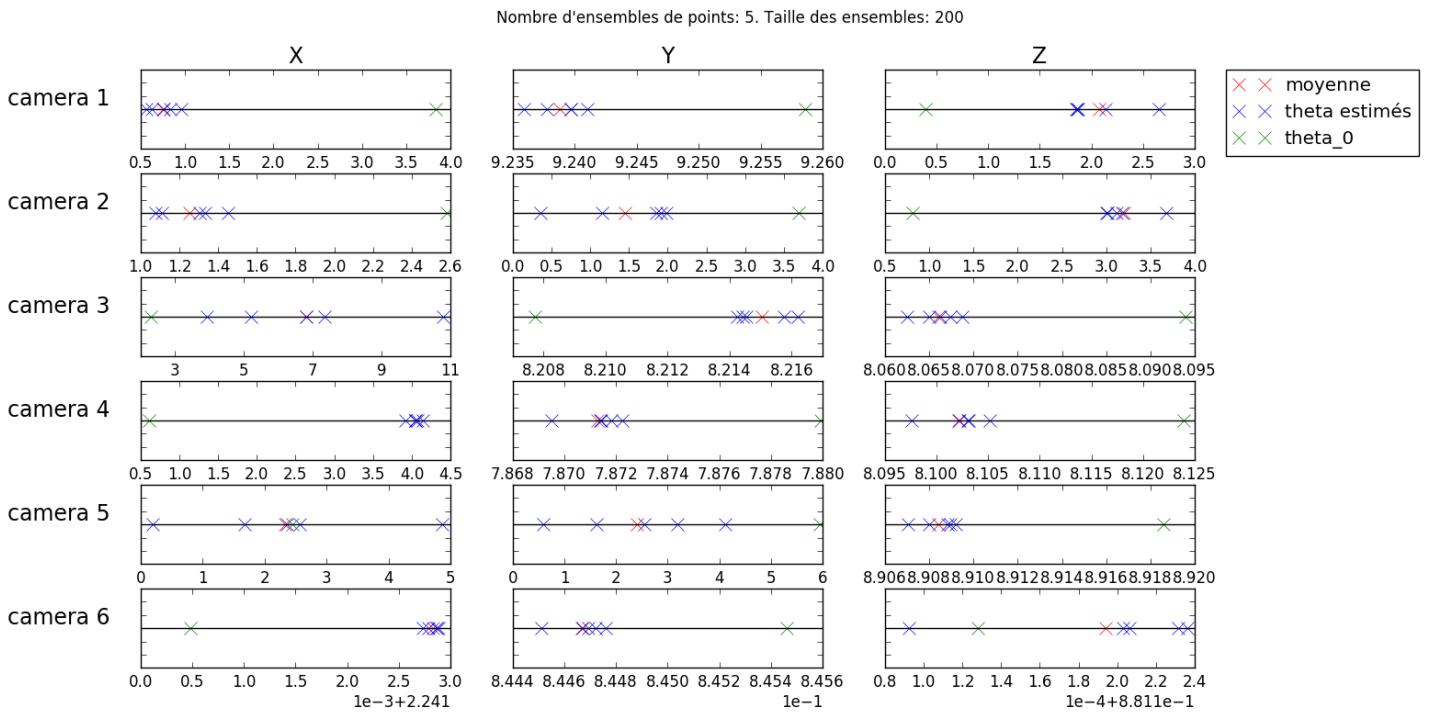


Figure 16: Distances des différentes solutions à la solution initiale selon θ

la somme des erreurs de reprojection c'est-à-dire minimise la fonction:

$$S(P, \mathbf{X}) = \sum_{i,j} (P_i \mathbf{X}^j - \hat{\mathbf{x}}_i^j)^2$$

Or notre étude se place dans le cas où les positions et les paramètres internes des caméras sont connues. Cela permet de se ramener à minimiser sur θ et sur $(\mathbf{X}^j)_{0 \leq j \leq N}$ la quantité:

$$S(\theta, \mathbf{X}) = \sum_{i,j} (P_i \mathbf{X}^j - \hat{\mathbf{x}}_i^j)^2$$

Pour des N peu grands (i.e. ≤ 80) nous avons fait des simulations en utilisant le solveur python de la bibliothèque `scipy.optimize` et avons comparé les solutions obtenues avec celles obtenues par la méthode des déterminants ainsi que les temps d'exécution.

Les solutions fournies par notre algorithme (déterminant) sont meilleures et plus régulières selon les angles.

[[2] ?]

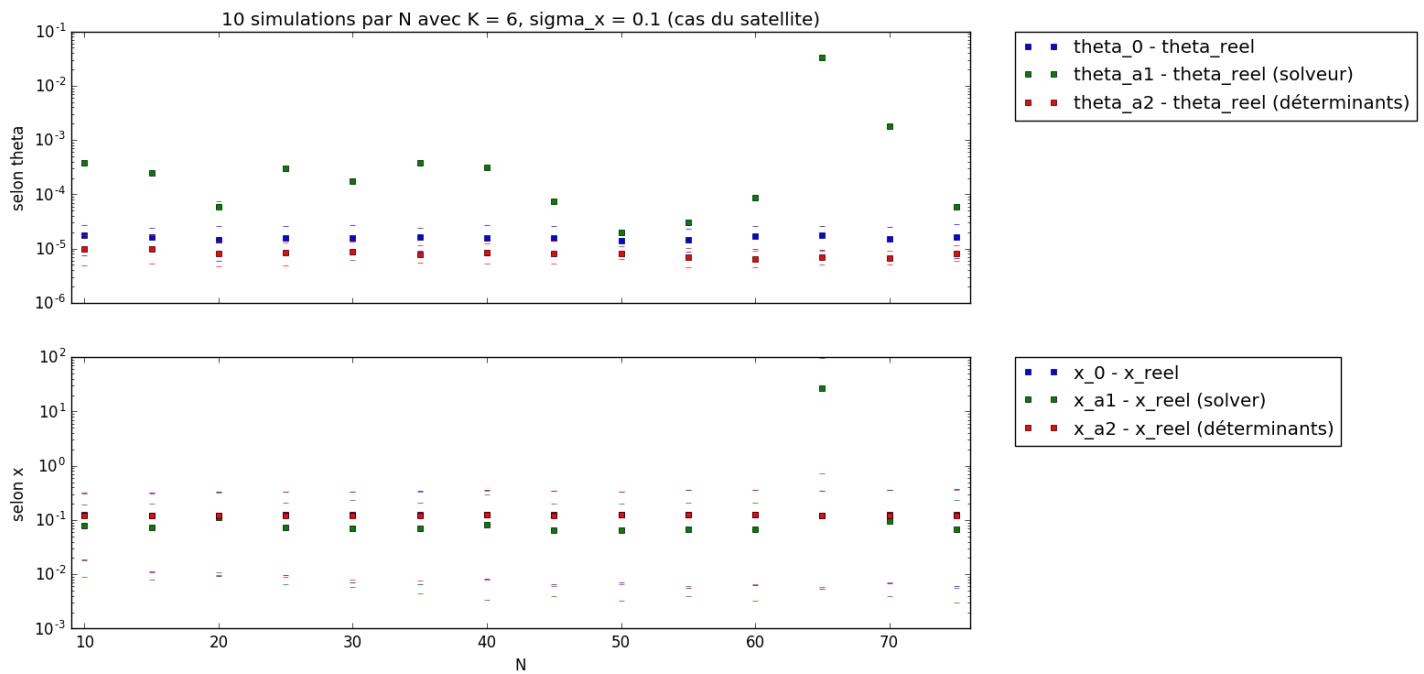


Figure 17: Comparaison des solutions de notre algorithme et du solveur python selon θ et selon x (moyenne sur 10 expériences pour chaque N)

References

- [1] Richard HARTLEY and Andrew ZISSEMAN. *Multiple view geometry in computer vision*. Cambridge University Press, 2003.
- [2] Pierre MOULON, Pascal MONASSE, Renaud MARLET, and Others. OpenMVG. <https://github.com/openMVG/openMVG>.

ANNEXES

L'ensemble des codes Python présentés dans ces annexes peut être retrouvé sur Github à l'adresse <https://github.com/Tooine/BA>.

ANNEXE A Fonctions de base

Les fonctions suivantes sont couramment utilisées pour chacune des simulations :

```

1 ## packages
2
3 import numpy as np
4 from mpl_toolkits.mplot3d import Axes3D
5 from numpy.random import rand, normal
6 import scipy.linalg as sl
7 from scipy.linalg import rq
8 from math import atan2
9
10 ## fonctions
11
12 def det(C1, C2, theta1, theta2, p1, p2, f):
13     """ retourne la valeur du déterminant associé à un point et deux caméras
14     et aux paramètres
15
16     PARAMETRES
17     C1, C2 [array (3,)] : coordonnées des centres des caméras ,
18     theta1, theta2 [array (3,)] : angles de rotation associés aux caméras ,
19     p1, p2 [array (2,)] : coordonnées des images des points sur les plans
19     images des caméras
20
21 """
22
23 ca1, ca2 = np.cos(theta1[0]), np.cos(theta2[0])
24 sa1, sa2 = np.sin(theta1[0]), np.sin(theta2[0])
25 cb1, cb2 = np.cos(theta1[1]), np.cos(theta2[1])
26 sb1, sb2 = np.sin(theta1[1]), np.sin(theta2[1])
27 cg1, cg2 = np.cos(theta1[2]), np.cos(theta2[2])
28 sg1, sg2 = np.sin(theta1[2]), np.sin(theta2[2])
29 x1, x2 = p1[0], p2[0]
30 y1, y2 = p1[1], p2[1]
31 m1, m2, m3 = C2[0]-C1[0], C2[1]-C1[1], C2[2]-C1[2]
32
33 return (
34     m1 * ((ca1*(-sg1*x1+cg1*y1) + sa1*(sb1*(cg1*x1+sg1*y1) + f*cb1)) * (sa2
34         *(sg2*x2-cg2*y2) + ca2*(sb2*(cg2*x2+sg2*y2) + f*cb2)) - (ca2*(-sg2*x2+cg2*y2) + sa2*(sb2*(cg2*x2+sg2*y2) + f*cb2)) * (sa1*(sg1*x1-cg1*y1) + ca1*(sb1*(cg1*x1+sg1*y1) + f*cb1)))

```

```

35      - m2 * ((cb1*(cg1*x1+sg1*y1) - f*sb1) * (sa2*(sg2*x2-cg2*y2) + ca2*(sb2
36          *(cg2*x2+sg2*y2) + f*cb2)) - (cb2*(cg2*x2+sg2*y2) - f*sb2) * (sa1*
37              sg1*x1-cg1*y1) + ca1*(sb1*(cg1*x1+sg1*y1) + f*cb1)))
38      + m3 * ((cb1*(cg1*x1+sg1*y1) - f*sb1) * (ca2*(-sg2*x2+cg2*y2) + sa2*
39          sb2*(cg2*x2+sg2*y2) + f*cb2)) - (cb2*(cg2*x2+sg2*y2) - f*sb2) * (
40              ca1*(-sg1*x1+cg1*y1) + sa1*(sb1*(cg1*x1+sg1*y1) + f*cb1)))
41      ) / f**2
42
43 def deriv_det(C1, C2, theta1, theta2, p1, p2, f):
44     """ Retourne les dérivées partielles du déterminant associé à un point
45     et deux caméras et aux paramètres
46
47     PARAMETRES
48     C1, C2 [array (3,)] : coordonnées des centres des caméras ,
49     theta1, theta2 [array (3,)] : angles de rotation associés aux caméras ,
50     p1, p2 [array (2,)] : coordonnées des images des points sur les plans
51         images des caméras
52
53     SORTIE
54     D [array (5, 2)] : retourne les dérivées partielles par rapport (dans l
55         'ordre par ligne) aux alpha_i , beta_i , gamma_i , x_i et y_i . (colonne
56         0 : point 1, colonne 1 : point 2)
57
58     """
59
60     # notations
61     ca1, ca2 = np.cos(theta1[0]), np.cos(theta2[0])
62     sa1, sa2 = np.sin(theta1[0]), np.sin(theta2[0])
63     cb1, cb2 = np.cos(theta1[1]), np.cos(theta2[1])
64     sb1, sb2 = np.sin(theta1[1]), np.sin(theta2[1])
65     cg1, cg2 = np.cos(theta1[2]), np.cos(theta2[2])
66     sg1, sg2 = np.sin(theta1[2]), np.sin(theta2[2])
67     x1, x2 = p1[0], p2[0]
68     y1, y2 = p1[1], p2[1]
69     m1, m2, m3 = C2[0]-C1[0], C2[1]-C1[1], C2[2]-C1[2]
70     fk1 = cb1*(cg1*x1+sg1*y1) - f*sb1
71     fl1 = cb2*(cg2*x2+sg2*y2) - f*sb2
72     fk2 = ca1*(-sg1*x1+cg1*y1) + sa1*(sb1*(cg1*x1+sg1*y1) + f*cb1)
73     fl2 = ca2*(-sg2*x2+cg2*y2) + sa2*(sb2*(cg2*x2+sg2*y2) + f*cb2)
74     fk3 = sa1*(sg1*x1-cg1*y1) + ca1*(sb1*(cg1*x1+sg1*y1) + f*cb1)
75     fl3 = sa2*(sg2*x2-cg2*y2) + ca2*(sb2*(cg2*x2+sg2*y2) + f*cb2)
76
77     # remplissage de la matrice D
78     D = np.zeros((5, 2))
79
80     D[0, 0] = (m1 * ((-sa1*(-sg1*x1+cg1*y1) + ca1*(sb1*(cg1*x1+sg1*y1) + f*
81         cb1)) * fl3 - fl2 * (ca1*(sg1*x1-cg1*y1) - sa1*(sb1*(cg1*x1+sg1*y1)
82             + f*cb1)))
83     - m2 * (- fl1 * (ca1*(sg1*x1-cg1*y1) - sa1*(sb1*(cg1*x1+sg1*y1) + f*cb1
84                 ))))

```

```

75 + m3 * (- fl1 * (-sa1*(-sg1*x1+cg1*y1) + ca1*(sb1*(cg1*x1+sg1*y1) + f*
    cb1)))) )
76
77 D[0, 1] = (m1 * (fk2 * (ca2*(sg2*x2-cg2*y2) - sa2*(sb2*(cg2*x2+sg2*y2)
    + f*cb2)) - (-sa2*(-sg2*x2+cg2*y2) + ca2*(sb2*(cg2*x2+sg2*y2) + f*
    cb2)) * fk3))
78 - m2 * (fk1 * (ca2*(sg2*x2-cg2*y2) - sa2*(sb2*(cg2*x2+sg2*y2) + f*cb2)))
    )
79 + m3 * (fk1 * (-sa2*(-sg2*x2+cg2*y2) + ca2*(sb2*(cg2*x2+sg2*y2) + f*cb2
    ))))
80
81 D[1, 0] = (m1 * (sa1*(cb1*(cg1*x1+sg1*y1) - f*sb1) * fl3 - fl2 * (ca1*(
    cb1*(cg1*x1+sg1*y1) - f*sb1)))
82 - m2 * ((-sb1*(cg1*x1+sg1*y1) - f*cb1) * fl3 - fl1 * (ca1*(cb1*(cg1*x1+
    sg1*y1) - f*sb1)))
83 + m3 * ((-sb1*(cg1*x1+sg1*y1) - f*cb1) * fl2 - fl1 * (sa1*(cb1*(cg1*x1+
    sg1*y1) - f*sb1))))
84
85 D[1, 1] = (m1 * (fk2 * (ca2*(cb2*(cg2*x2+sg2*y2) - f*sb2)) - (sa2*(cb2
    *(cg2*x2+sg2*y2) - f*sb2)) * fk3))
86 - m2 * (fk1 * (ca2*(cb2*(cg2*x2+sg2*y2) - f*sb2)) - (-sb2*(cg2*x2+sg2*
    y2) - f*cb2) * fk3)
87 + m3 * (fk1 * (sa2*(cb2*(cg2*x2+sg2*y2) - f*sb2)) - (-sb2*(cg2*x2+sg2*
    y2) - f*cb2) * fk2))
88
89 D[2, 0] = (m1 * ((ca1*(-cg1*x1-sg1*y1) + sa1*sb1*(-sg1*x1+cg1*y1)) *
    fl3 - fl2 * (sa1*(cg1*x1+sg1*y1) + ca1*sb1*(-sg1*x1+cg1*y1)))
90 - m2 * (cb1*(-sg1*x1+cg1*y1) * fl3 - fl1 * (sa1*(cg1*x1+sg1*y1) + ca1*
    sb1*(-sg1*x1+cg1*y1)))
91 + m3 * (cb1*(-sg1*x1+cg1*y1) * fl2 - fl1 * (ca1*(-cg1*x1-sg1*y1) + sa1*
    sb1*(-sg1*x1+cg1*y1))))
92
93 D[2, 1] = (m1 * (fk2 * (sa2*(cg2*x2+sg2*y2) + ca2*sb2*(-sg2*x2+cg2*y2))
    - (ca2*(-cg2*x2-sg2*y2) + sa2*sb2*(-sg2*x2+cg2*y2)) * fk3))
94 - m2 * (fk1 * (sa2*(cg2*x2+sg2*y2) + ca2*sb2*(-sg2*x2+cg2*y2)) - cb2*(-
    sg2*x2+cg2*y2) * fk3)
95 + m3 * (fk1 * (ca2*(-cg2*x2-sg2*y2) + sa2*sb2*(-sg2*x2+cg2*y2)) - cb2
    *(-sg2*x2+cg2*y2) * fk2))
96
97 D[3, 0] = (m1 * ((ca1*-sg1 + sa1*sb1*cg1) * fl3 - fl2 * (sa1*sg1 + ca1*
    sb1*cg1)))
98 - m2 * (cb1*cg1 * fl3 - fl1 * (sa1*sg1 + ca1*sb1*cg1))
99 + m3 * (cb1*cg1 * fl2 - fl1 * (ca1*-sg1 + sa1*sb1*cg1)))
100
101 D[3, 1] = (m1 * (fk2 * (sa2*sg2 + ca2*sb2*cg2) - (ca2*-sg2 + sa2*sb2*
    cg2) * fk3))
102 - m2 * (fk1 * (sa2*sg2 + ca2*sb2*cg2) - cb2*cg2 * fk3)
103 + m3 * (fk1 * (ca2*-sg2 + sa2*sb2*cg2) - cb2*cg2 * fk2))
104
105 D[4, 0] = (m1 * ((ca1*cg1 + sa1*sb1*sg1) * fl3 - fl2 * (sa1*-cg1 + ca1*
    sb1*sg1)))

```

```

          sb1 * sg1))
106 - m2 * (cb1 * sg1 * fl3 - fl1 * (sa1 * -cg1 + ca1 * sb1 * sg1))
107 + m3 * (cb1 * sg1 * fl2 - fl1 * (ca1 * cg1 + sa1 * sb1 * sg1)))
108
109 D[4, 1] = (m1 * (fk2 * (sa2 * -cg2 + ca2 * sb2 * sg2) - (ca2 * cg2 + sa2 * sb2 *
110     sg2) * fk3)
111 - m2 * (fk1 * (sa2 * -cg2 + ca2 * sb2 * sg2) - cb2 * sg2 * fk3)
112 + m3 * (fk1 * (ca2 * cg2 + sa2 * sb2 * sg2) - cb2 * sg2 * fk2))
113
114 return D / f ** 2
115
116 def genere_liste_couples(K):
117     """ crée la liste des couples (i1, i2) pour i1 < i2 entre 0 et K-1 """
118
119     liste_couples = []
120     for i1 in range(K-1):
121         for i2 in range(i1+1, K):
122             liste_couples.append((i1, i2))
123
124 return liste_couples
125
126 def matrices_Aj_Bj(C, theta, x, j, f, liste_couples):
127     """ Calcule simultanément des matrices Aj et Bj associées au j-ième point
128     de x
129
130     PARAMETRES
131     C [array (K, 3)] : ensemble des coordonnées des K caméras
132     theta [array (K, 3)] : ensemble des angles de rotations des caméras
133     x [array (N, K, 2)] : ensemble des coordonnées des images sur les K caméras
134     des N points
135
136     """
137
138     K = C.shape[0]
139     Aj, Bj = np.zeros((K*(K-1)//2, 3*K)), np.zeros((K*(K-1)//2, 2*K))
140     for l, (i1, i2) in enumerate(liste_couples):
141         derivees = deriv_det(C[i1], C[i2], theta[i1], theta[i2], x[j, i1],
142                               x[j, i2], f)
143         Aj[l, 3*i1:3*i1+3] = derivees[:3, 0]
144         Aj[l, 3*i2:3*i2+3] = derivees[:3, 1]
145         Bj[l, 2*i1:2*i1+2] = derivees[3:, 0]
146         Bj[l, 2*i2:2*i2+2] = derivees[3:, 1]
147
148     return Aj, Bj
149
150 def matrice_Aj(C, theta, x, j, f, liste_couples):
151     """ Calcule la matrice Aj associée au j-ième point de x
152
153     PARAMETRES
154     C [array (K, 3)] : ensemble des coordonnées des K caméras

```

```

151     theta [array (K, 3)] : ensemble des angles de rotations des caméras
152     x [array (N, K, 2)]: ensemble des coordonnées des images sur les K camé-
153     ras des N points
154
155
156     K = C.shape[0]
157     Aj = np.zeros((K*(K-1)//2 , 3*K))
158     for (l, (i1, i2)) in enumerate(liste_couples):
159         derivees = deriv_det(C[i1], C[i2], theta[i1], theta[i2], x[j, i1],
160                               x[j, i2], f)
161         Aj[l, 3*i1:3*i1+3] = derivees[:3, 0]
162         Aj[l, 3*i2:3*i2+3] = derivees[:3, 1]
163
164     return Aj
165
166 def matrice_Bj(C, theta, x, j, f, liste_couples):
167     """ Calcule la matrice Bj associée au j-ième point de x
168
169     PARAMETRES
170     C [array (K, 3)] : ensemble des coordonnées des K caméras
171     theta [array (K, 3)] : ensemble des angles de rotations des caméras
172     x [array (N, K, 2)]: ensemble des coordonnées des images sur les K camé-
173     ras des N points
174
175     K = C.shape[0]
176     Bj = np.zeros((K*(K-1)//2 , 2*K))
177     for (l, (i1, i2)) in enumerate(liste_couples):
178         derivees = deriv_det(C[i1], C[i2], theta[i1], theta[i2], x[j, i1],
179                               x[j, i2], f)
180         Bj[l, 2*i1:2*i1+2] = derivees[3:, 0]
181         Bj[l, 2*i2:2*i2+2] = derivees[3:, 1]
182
183     return Bj
184
185 def matrices_A_B(C, theta, x, f):
186     """ Calcule les matrices A et B
187
188     PARAMETRES
189     C [array (K, 3)] : ensemble des coordonnées des K caméras
190     theta [array (K, 3)] : ensemble des angles de rotations des caméras
191     x [array (N, K, 2)]: ensemble des coordonnées des images sur les K cam-
192     éras des N points
193
194     K, N = C.shape[0], x.shape[0]
195     A, B = np.zeros((K*(K-1)//2*N, 3*K)), np.zeros((K*(K-1)//2*N, 2*K*N))

```

```

196     for j in range(N):
197         A[K*(K-1)//2*j : K*(K-1)//2*(j+1)], B[K*(K-1)//2*j : K*(K-1)//2*(j+1),
198             2*K*j : 2*K*(j+1)] = matrices_Aj_Bj(C, theta, x, j, f,
199             genere_liste_couples(K))
200
201     return A, B
202
203 def matrice_A(C, theta, x, f):
204     """ Calcule la matrice A
205
206     PARAMETRES
207     C [array (K, 3)] : ensemble des coordonnées des K caméras
208     theta [array (K, 3)] : ensemble des angles de rotations des caméras
209     x [array (N, K, 2)]: ensemble des coordonnées des images sur les K caméras des N points
210
211     """
212
213     K, N = C.shape[0], x.shape[0]
214     A = np.zeros((K*(K-1)//2*N, 3*K))
215     for j in range(N):
216         A[K*(K-1)//2*j : K*(K-1)//2*(j+1)] = matrice_Aj(C, theta, x, j, f,
217             genere_liste_couples(K))
218
219     return A
220
221 def matrice_B(C, theta, x, f):
222     """ Calcule la matrice B
223
224     PARAMETRES
225     C [array (K, 3)] : ensemble des coordonnées des K caméras
226     theta [array (K, 3)] : ensemble des angles de rotations des caméras
227     x [array (N, K, 2)]: ensemble des coordonnées des images sur les K caméras des N points
228
229     """
230
231     K, N = C.shape[0], x.shape[0]
232     B = np.zeros((K*(K-1)//2*N, 2*K*N))
233     for j in range(N):
234         B[K*(K-1)//2*j : K*(K-1)//2*(j+1), 2*K*j : 2*K*(j+1)] = matrice_Bj(C,
235             theta, x, j, f, genere_liste_couples(K))
236
237     return B
238
239 def F(C, theta, x, f):
240     """ Calcule les valeurs de tous les déterminants du problème
241
242     PARAMETRES
243     C [array (K, 3)] : ensemble des coordonnées des K caméras

```

```

240     theta [array (K, 3)] : ensemble des angles de rotations des caméras
241     x [array (N, K, 2)]: ensemble des coordonnées des images sur les K camé-
242     ras des N points
243
244     SORTIE
245     FF [list (N*K*(K-1)//2)]: valeurs de tous les déterminants dans l'
246     ordre de nos conventions
247
248     """
249
250     N, K = x.shape[:-1]
251     FF = []
252     for j in range(N):
253         for (i1, i2) in genere_liste_couples(K):
254             FF.append(det(C[i1], C[i2], theta[i1], theta[i2], x[j, i1], x[j,
255             , i2], f))
256
257     return np.array(FF)
258
259 def RX(t):
260     """ matrice de rotation selon X d'angle t """
261
262     return(np.array([[1, 0, 0],
263                     [0, np.cos(t), -np.sin(t)],
264                     [0, np.sin(t), np.cos(t)]]))
265
266 def RY(t):
267     """ matrice de rotation selon Y d'angle t """
268
269     return(np.array([[np.cos(t), 0, np.sin(t)],
270                     [0, 1, 0],
271                     [-np.sin(t), 0, np.cos(t)]]))
272
273 def RZ(t):
274     """ matrice de rotation selon Z d'angle t """
275
276     return(np.array([[np.cos(t), -np.sin(t), 0],
277                     [np.sin(t), np.cos(t), 0],
278                     [0, 0, 1]]))
279
280 def matrice_R(theta):
281     """ matrice de rotation de la forme Rz Ry Rx
282
283     PARAMETRE
284     theta [array (3,)] : angles de rotations
285
286     """
287
288     return np.dot(RZ(theta[2]), np.dot(RY(theta[1]), RX(theta[0])))

```

```

287 def matrice_P(C, theta, f):
288     """ crée la matrice de projection d'une caméra
289
290     PARAMETRES
291     C [array (3,)] : coordonnées de la caméra
292     theta [array (3,)] : angles de rotations de la caméra
293     f : focale en pixels
294
295     """
296
297     R = matrice_R(theta)
298     K = np.array([[f, 0, 0], [0, f, 0], [0, 0, 1]])
299     P = np.zeros((3, 4))
300     P[:, :3] = np.dot(K, R)
301     P[:, 3] = np.dot(K, R).dot(-C)
302
303     return P
304
305 def kr_from_p(P):
306     """ Extract K, R and C from a camera matrix P, such that P = K*R*[eye
307         (3) / -C].
308
309     K is scaled so that K[2, 2] = 1 and K[0, 0] > 0.
310
311     """
312     K, R = rq(P[:, :3])
313
314     K /= K[2, 2]
315     if K[0, 0] < 0:
316         D = np.diag([-1, -1, 1])
317         K = np.dot(K, D)
318         R = np.dot(D, R)
319
320     C = -np.linalg.solve(P[:, :3], P[:, 3])
321
322     test = np.dot(np.dot(K, R), np.concatenate((np.eye(3), -np.array([C]).T),
323                                                 axis=1))
324     np.testing.assert_allclose(test / test[2, 3], P / P[2, 3])
325
326     return C, R, K
327
328 def theta_from_r(R):
329     a = atan2(R[2, 1], R[2, 2])
330     b = atan2(-R[2, 0], np.sqrt(R[2, 1]**2 + R[2, 2]**2))
331     g = atan2(R[1, 0], R[0, 0])
332
333     return a, b, g

```

ANNEXE B Simulations sans données réelles

On a besoin des packages suivants :

```

1 ## packages
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5 from mpl_toolkits.mplot3d import Axes3D
6 from numpy.random import rand, normal
7 import scipy.linalg as sl
8 from scipy.sparse import csr_matrix
9 import scipy.sparse.linalg as ssl
10 import os
11
12 plt.ion(), plt.show()
13
14 ## import de BA_fonctions
15
16 dossier = "/Users/abarrier/Documents/Scolaire/ENS/L3/Cours/Stage/Python"
17
18 os.chdir(dossier)
19 from BA_fonctions import *

```

Les 6 fonctions suivantes permettent de faire toute les simulations sans terme forçant nécessaires :

```

1 ## rappels
2
3 # on suppose les points de correspondances situés dans la zone
4 # [-X0/2, X0/2] x [-Y0/2, Y0/2] x [-Z0/2, Z0/2]
5
6 # on suppose les caméras situées dans la zone
7 # [-X0, X0] x [-Y0, Y0] x [Z1, Z2] avec Z0 << Z1
8
9 # on suppose que les caméras pointent vers le point (0, 0, 0)
10 # pour cela, connaissant les coordonnées de la caméra, on fixe l'angle
11 # gamma à 0
12 # puis alpha et beta sont alors déterminés de manière unique
13
14 ## fonctions
15
16 def genere_donnees():
17     """ crée aléatoirement des données
18
19     SORTIES
20     C_reel [array (K, 3)] : coordonnées des caméras
21     X_reel [array (N, 3)] : coordonnées des points
22     theta_reel [array (K, 3)] : angles de rotation des caméras initiaux
23
24     """

```

```

25 C_reel = rand(K, 3)
26 C_reel[:, 0] = (C_reel[:, 0] - 0.5) * 2 * X0
27 C_reel[:, 1] *= (C_reel[:, 1] - 0.5) * 2 * Y0
28 C_reel[:, 2] = C_reel[:, 2] * (Z2-Z1) + Z1
29
30 theta_reel = np.zeros((K, 3))
31 for i in range(K): # on fait pointer les caméras vers le point (0, 0,
32     0)
33     X, Y, Z = -C_reel[i] / sl.norm(C_reel[i])
34     beta = np.arcsin(-X)
35     alpha = np.arcsin(Y / np.cos(beta))
36     if np.cos(alpha) * np.cos(beta) * Z < 0: alpha = np.pi - alpha
37     theta_reel[i, 0] = alpha
38     theta_reel[i, 1] = beta
39
40 X_reel = (rand(N, 3) - 0.5)
41 X_reel[:, 0] *= X0
42 X_reel[:, 1] *= Y0
43 X_reel[:, 2] *= Z0
44
45 return C_reel, X_reel, theta_reel
46
47 def scene_3D():
48     """ crée une figure 3D correspondant aux données """
49
50     Z_cam = []
51     for theta in theta_reel:
52         Z_cam.append(np.linalg.inv(matrice_R(theta)).dot(np.array([0, 0,
53             1])))
54
55     plt.figure().gca(projection = "3d")
56     plt.xlabel("X"), plt.ylabel("Y")
57     plt.plot([0], [0], [0], color="k", marker="+", markersize=10)
58     plt.plot(X_reel[:, 0], X_reel[:, 1], X_reel[:, 2], marker="+",
59             markersize=3
60             , color="b", label="points", linestyle="None")
61     plt.plot(C_reel[:, 0], C_reel[:, 1], C_reel[:, 2], marker="+",
62             markersize=10
63             , color="r", label="cameras", linestyle="None")
64     plt.legend(loc="best")
65     plt.xlim(-X0, X0), plt.ylim(-Y0, Y0)
66
67     t = 200000
68     for C, Z in zip(C_reel, Z_cam): # trace les axes Z_cam
69         ZZ = C + t * Z
70         plt.plot([C[0], ZZ[0]], [C[1], ZZ[1]], [C[2], ZZ[2]], color="k")
71
72 def reestimation_globale(x_exact=False, theta_exact=False, affichage=False):
73     """ réalise une simulation globale (création des données + réestimation

```

70 de tous les paramètres)

71 **PARAMETRES**

72 x_exact [bool] : si vrai, fixe x_0 à x_{reel} (pas de perturbation des points)

73 theta_exact [bool] : si vrai, fixe θ_0 à θ_{reel} (pas de perturbation des angles)

74 affichage [bool] : si vrai, affiche les données créées sur une figure 3D

75 **SORTIES**

76 C_reel [array (K, 3)] : coordonnées des caméras

77 X_reel [array (N, 3)] : coordonnées des points

78 theta_reel, theta_0, theta_a [array (K, 3)] : angles de rotation des caméras, respectivement initiaux, perturbés et réestimés

79 x_reel, x_0, x_a [array (N, K, 2)] : coordonnées des images des points, respectivement initiaux, perturbés et réestimés

80 """

81 C_reel, X_reel, theta_reel = genere_donnees()

82 **if** affichage: scene_3D()

83 matrices_P = []

84 **for** i **in** range(K):

85 matrices_P.append(matrice_P(C_reel[i], theta_reel[i], f))

86 x_reel = np.zeros((N, K, 2))

87 **for** j **in** range(N):

88 **for** i **in** range(K):

89 point = matrices_P[i].dot(np.array([X_reel[j, 0], X_reel[j, 1], X_reel[j, 2], 1]))

90 x_reel[j, i] = point[:-1] / point[-1]

91 # perturbation des coordonnées et des angles

92 x_0 = np.copy(x_reel)

93 **if** not x_exact: x_0 += normal(0, sigma_x, (N, K, 2))

94 theta_0 = np.copy(theta_reel)

95 **if** not theta_exact: theta_0 += normal(0, sigma_theta, (K, 3))

96 # minimisation de F

97 A, B = matrices_A_B(C_reel, theta_0, x_0, f)

98 M = csr_matrix(np.concatenate((A, B), axis=1))

99 erreur = ssl.lsqr(M, -F(C_reel, theta_0, x_0, f), conlim=1.0e+8)[0]

100 theta_a = theta_0 + erreur[:3*K].reshape((K, 3))

```

113     x_a = np.copy(x_0)
114     for j in range(N):
115         for i in range(K):
116             x_a[j, i, 0] += erreur[3*K + 2*K*j + 2*i]
117             x_a[j, i, 1] += erreur[3*K + 2*K*j + 2*i + 1]
118
119     return C_reel, X_reel, theta_reel, theta_0, theta_a, x_reel, x_0, x_a
120
121 def reestimation_x(theta_exact=False, affichage=False):
122     """ réalise une simulation ne réestimant que les x (création des données + réestimation des x)
123
124     PARAMETRES
125     x_exact [bool] : si vrai, fixe x_0 à x_reel (pas de perturbation des points)
126     theta_exact [bool] : si vrai, fixe theta_0 à theta_reel (pas de perturbation des angles)
127     affichage [bool] : si vrai, affiche les données créées sur une figure 3D
128
129     SORTIES
130     C_reel [array (K, 3)] : coordonnées des caméras
131     X_reel [array (N, 3)] : coordonnées des points
132     theta_reel, theta_0, theta_a [array (K, 3)] : angles de rotation des caméras, respectivement initiaux, perturbés et réestimés
133     x_reel, x_0, x_a [array (N, K, 2)] : coordonnées des images des points, respectivement initiaux, perturbés et réestimés
134
135 """
136
137     C_reel, X_reel, theta_reel = genere_donnees()
138
139     if affichage: scene_3D()
140
141     matrices_P = []
142     for i in range(K):
143         matrices_P.append(matrice_P(C_reel[i], theta_reel[i], f))
144
145     x_reel = np.zeros((N, K, 2))
146     for j in range(N):
147         for i in range(K):
148             point = matrices_P[i].dot(np.array([X_reel[j, 0], X_reel[j, 1],
149                                         X_reel[j, 2], 1]))
150             x_reel[j, i] = point[:-1] / point[-1]
151
152     # perturbation des coordonnées et des angles
153
154     x_0 = x_reel + normal(0, sigma_x, (N, K, 2))
155     theta_0 = np.copy(theta_reel)
156     if not theta_exact: theta_0 += normal(0, sigma_theta, (K, 3))

```

```

156
157     # minimisation de F
158
159     B = csr_matrix(matrice_B(C_reel, theta_0, x_0, f))
160
161     erreur = ssl.lsqr(B, -F(C_reel, theta_0, x_0, f), conlim=1.0e+8)[0]
162
163     theta_a = np.copy(theta_0)
164     x_a = np.copy(x_0)
165     for j in range(N):
166         for i in range(K):
167             x_a[j, i, 0] += erreur[2*K*j + 2*i]
168             x_a[j, i, 1] += erreur[2*K*j + 2*i + 1]
169
170     return C_reel, X_reel, theta_reel, theta_0, theta_a, x_reel, x_0, x_a
171
172 def reestimation_theta(x_exact=False, affichage=False):
173     """ réalise une simulation ne réestimant que les angles (création des
174     données + réestimation des angles)
175
176     PARAMETRES
177     x_exact [bool] : si vrai, fixe x_0 à x_reel (pas de perturbation des
178     points)
179     theta_exact [bool] : si vrai, fixe theta_0 à theta_reel (pas de
180     perturbation des angles)
181     affichage [bool] : si vrai, affiche les données créées sur une figure 3
182     D
183
184     SORTIES
185     C_reel [array (K, 3)] : coordonnées des caméras
186     X_reel [array (N, 3)] : coordonnées des points
187     theta_reel, theta_0, theta_a [array (K, 3)] : angles de rotation des
188     caméras, respectivement initiaux, perturbés et réestimés
189     x_reel, x_0, x_a [array (N, K, 2)] : coordonnées des images des points,
190     respectivement initiaux, perturbés et réestimés
191
192     """
193
194     C_reel, X_reel, theta_reel = genere_donnees()
195
196     if affichage: scene_3D()
197
198     matrices_P = []
199     for i in range(K):
200         matrices_P.append(matrice_P(C_reel[i], theta_reel[i], f))
201
202     x_reel = np.zeros((N, K, 2))
203     for j in range(N):
204         for i in range(K):
205             point = matrices_P[i].dot(np.array([X_reel[j, 0], X_reel[j, 1],
206
207

```

```

200         X_reel[j, 2], 1)))
201     x_reel[j, i] = point[:-1] / point[-1]
202
203     # perturbation des coordonnées et des angles
204
205     x_0 = np.copy(x_reel)
206     if not x_exact: x_0 += normal(0, sigma_x, (N, K, 2))
207     theta_0 = theta_reel + normal(0, sigma_theta, (K, 3))
208
209     # minimisation de F
210
211     A = csr_matrix(matrice_A(C_reel, theta_0, x_0, f))
212
213     erreur = ssl.lsqr(A, -F(C_reel, theta_0, x_0, f), conlim=1.0e+8)[0]
214
215     theta_a = theta_0 + erreur.reshape((K, 3))
216     x_a = np.copy(x_0)
217
218     return C_reel, X_reel, theta_reel, theta_0, theta_a, x_reel, x_0, x_a
219
220 def reestimation_parametres_choisis(x_fixes, theta_fixes, affichage=False):
221     """ réalise une simulation qui ne réestime que les paramètres souhaités
222
223     PARAMETRES
224     x_fixes [array (N, K, 2)] : array de 0 et de 1. Les 1 correspondent aux
225     coordonnées qu'il ne faut pas réestimer
226     theta_fixes [array (K, 3)] : array de 0 et de 1. Les 1 correspondent
227     aux angles qu'il ne faut pas réestimer
228     affichage [bool] : si vrai, affiche les données créées sur une figure 3
229     D
230
231     SORTIES
232     C_reel [array (K, 3)] : coordonnées des caméras
233     X_reel [array (N, 3)] : coordonnées des points
234     theta_reel, theta_0, theta_a [array (K, 3)] : angles de rotation des
235     caméras, respectivement initiaux, perturbés et réestimés
236     x_reel, x_0, x_a [array (N, K, 2)] : coordonnées des images des points,
237     respectivement initiaux, perturbés et réestimés
238
239     """
240
241     C_reel, X_reel, theta_reel = genere_donnees()
242
243     if affichage: scene_3D()
244
245     matrices_P = []
246     for i in range(K):
247         matrices_P.append(matrice_P(C_reel[i], theta_reel[i], f))
248
249     x_reel = np.zeros((N, K, 2))

```

```

244     for j in range(N):
245         for i in range(K):
246             point = matrices_P[i].dot(np.array([X_reel[j, 0], X_reel[j, 1],
247                                         X_reel[j, 2], 1]))
248             x_reel[j, i] = point[:-1] / point[-1]
249
250 # perturbation des coordonnées et des angles
251
252 x_0 = x_reel + normal(0, sigma_x, (N, K, 2))
253 theta_0 = theta_reel + normal(0, sigma_theta, (K, 3))
254
255 for n in range(N):
256     for k in range(K):
257         for i in range(2):
258             if x_fixes[n, k, i] == 1:
259                 x_0[n, k, i] = x_reel[n, k, i]
260
261 for k in range(K):
262     for angle in range(3):
263         if theta_fixes[k, angle] == 1:
264             theta_0[k, angle] = theta_reel[k, angle]
265
266 # minimisation de F
267
268 A, B = matrices_A_B(C_reel, theta_0, x_0, f)
269 for k in range(K-1, -1, -1):
270     for angle in range(2, -1, -1):
271         if theta_fixes[k, angle] == 1:
272             A = np.delete(A, (3*k+angle), axis=1)
273
274 for n in range(N-1, -1, -1):
275     for k in range(K-1, -1, -1):
276         for i in range(1, -1, -1):
277             if x_fixes[n, k, i] == 1:
278                 B = np.delete(B, (2*K*n + 2*k + i), axis=1)
279
280 M = np.concatenate((A, B), axis=1)
281
282 erreur1 = ssl.lsqr(M, -F(C_reel, theta_0, x_0, f), conlim=1.0e+8)[0]
283
284 erreur = np.zeros((3*K + 2*N*K))
285 i = 0
286
287 for p in range(3*K):
288     if theta_fixes[p//3, p%3] == 0:
289         erreur[p] = erreur1[i]
290         i += 1
291
292 for p in range(3*K, 3*K + 2*N*K):
293     q = p - 3*K
294     if x_fixes[q//(2*K), (q%(2*K))//2, q%2] == 0:
295         erreur[p] = erreur1[i]
296         i += 1

```

```

293     theta_a = theta_0 + erreur[:3*K].reshape((K, 3))
294     x_a = np.copy(x_0)
295     for j in range(N):
296         for i in range(K):
297             x_a[j, i, 0] += erreur[3*K + 2*K*j + 2*i]
298             x_a[j, i, 1] += erreur[3*K + 2*K*j + 2*i + 1]
299
300     return C_reel, X_reel, theta_reel, theta_0, theta_a, x_reel, x_0, x_a

```

Et celles-ci permettent de faire les simulations avec un terme forçant :

```

1 def reestimation_globale_forcant(x_exact=False, theta_exact=False,
2                                   affichage=False):
3     """ réalise une simulation globale (création des données + réestimation
4     de tous les paramètres)
5
6     PARAMETRES
7     x_exact [bool] : si vrai, fixe x_0 à x_reel (pas de perturbation des
8     points)
9     theta_exact [bool] : si vrai, fixe theta_0 à theta_reel (pas de
10    perturbation des angles)
11    affichage [bool] : si vrai, affiche les données créées sur une figure 3
12    D
13
14    SORTIES
15    C_reel [array (K, 3)] : coordonnées des caméras
16    X_reel [array (N, 3)] : coordonnées des points
17    theta_reel, theta_0, theta_a [array (K, 3)] : angles de rotation des
18    caméras, respectivement initiaux, perturbés et réestimés
19    x_reel, x_0, x_a [array (N, K, 2)] : coordonnées des images des points,
20    respectivement initiaux, perturbés et réestimés
21
22    """
23
24
25    C_reel, X_reel, theta_reel = genere_donnees()
26
27    if affichage: scene_3D()
28
29    matrices_P = []
30    for i in range(K):
31        matrices_P.append(matrice_P(C_reel[i], theta_reel[i], f))
32
33    x_reel = np.zeros((N, K, 2))
34    for j in range(N):
35        for i in range(K):
36            point = matrices_P[i].dot(np.array([X_reel[j, 0], X_reel[j, 1],
37                                              X_reel[j, 2], 1]))
38            x_reel[j, i] = point[:-1] / point[-1]
39
40    # perturbation des coordonnées et des angles

```

```

33     x_0 = np.copy(x_reel)
34     if not x_exact: x_0 += normal(0, sigma_x, (N, K, 2))
35     theta_0 = np.copy(theta_reel)
36     if not theta_exact: theta_0 += normal(0, sigma_theta, (K, 3))
37
38     # minimisation de F
39
40     A, B = matrices_A_B(C_reel, theta_0, x_0, f)
41     M = np.concatenate((A, B), axis=1)
42
43     # ajout du terme forcant sur la matrice M
44     lamb = np.zeros(3*K+2*K*N)
45     lamb[0:3*K] = lamb_theta
46     lamb[3*K:] = lamb_x
47     sig = np.zeros(3*K+2*K*N)
48     sig[0:3*K] = sigma_theta
49     sig[3*K:] = sigma_x
50
51     contrainte = np.eye(3*K+2*N*K) * lamb / sig
52     M = csr_matrix(np.append(M, contrainte, axis=0))
53
54     b = np.zeros((K*(K-1)//2*N+3*K+2*N*K))
55     b[:K*(K-1)//2*N] = -F(C_reel, theta_0, x_0, f)
56
57     erreur = ssl.lsqr(M, b, conlim=1.0e+8)[0]
58
59     theta_a = theta_0 + erreur[:3*K].reshape((K, 3))
60     x_a = np.copy(x_0)
61     for j in range(N):
62         for i in range(K):
63             x_a[j, i, 0] += erreur[3*K + 2*K*j + 2*i]
64             x_a[j, i, 1] += erreur[3*K + 2*K*j + 2*i + 1]
65
66     return C_reel, X_reel, theta_reel, theta_0, theta_a, x_reel, x_0, x_a
67
68 def reestimation_x_forcant(theta_exact=False, affichage=False):
69     """ réalise une simulation ne réestimant que les x (création des données + réestimation des x) """
70
71     PARAMETRES
72     x_exact [bool] : si vrai, fixe x_0 à x_reel (pas de perturbation des points)
73     theta_exact [bool] : si vrai, fixe theta_0 à theta_reel (pas de perturbation des angles)
74     affichage [bool] : si vrai, affiche les données créées sur une figure 3D
75
76     SORTIES
77     C_reel [array (K, 3)] : coordonnées des caméras
78     X_reel [array (N, 3)] : coordonnées des points

```

```

79     theta_reel, theta_0, theta_a [array (K, 3)] : angles de rotation des
80         caméras, respectivement initiaux, perturbés et réestimés
81     x_reel, x_0, x_a [array (N, K, 2)] : coordonnées des images des points ,
82         respectivement initiaux, perturbés et réestimés
83
84     """
85
86     C_reel, X_reel, theta_reel = genere_donnees()
87
88     if affichage: scene_3D()
89
90     matrices_P = []
91     for i in range(K):
92         matrices_P.append(matrice_P(C_reel[i], theta_reel[i], f))
93
94     x_reel = np.zeros((N, K, 2))
95     for j in range(N):
96         for i in range(K):
97             point = matrices_P[i].dot(np.array([X_reel[j, 0], X_reel[j, 1],
98                                                 X_reel[j, 2], 1]))
99             x_reel[j, i] = point[:-1] / point[-1]
100
101     # perturbation des coordonnées et des angles
102
103     x_0 = x_reel + normal(0, sigma_x, (N, K, 2))
104     theta_0 = np.copy(theta_reel)
105     if not theta_exact: theta_0 += normal(0, sigma_theta, (K, 3))
106
107     # minimisation de F
108
109     B = matrice_B(C_reel, theta_0, x_0, f)
110
111     # ajout du terme forcant sur la matrice M
112     contrainte = np.eye(2*N*K) * lamb_x/sigma_x
113     M = csr_matrix(np.append(B, contrainte, axis=0))
114
115     erreur = ssl.lsqr(B, -F(C_reel, theta_0, x_0, f), conlim=1.0e+8)[0]
116
117     theta_a = np.copy(theta_0)
118     x_a = np.copy(x_0)
119     for j in range(N):
120         for i in range(K):
121             x_a[j, i, 0] += erreur[2*K*j + 2*i]
122             x_a[j, i, 1] += erreur[2*K*j + 2*i + 1]
123
124     return C_reel, X_reel, theta_reel, theta_0, theta_a, x_reel, x_0, x_a
125

```

```

126 def reestimation_theta_forcant(x_exact=False, affichage=False):
127     """ réalise une simulation ne réestimant que les angles (création des
128     données + réestimation des angles)
129
130     PARAMETRES
131     x_exact [bool] : si vrai, fixe x_0 à x_reel (pas de perturbation des
132     points)
133     theta_exact [bool] : si vrai, fixe theta_0 à theta_reel (pas de
134     perturbation des angles)
135     affichage [bool] : si vrai, affiche les données créées sur une figure 3
136     D
137
138     SORTIES
139     C_reel [array (K, 3)] : coordonnées des caméras
140     X_reel [array (N, 3)] : coordonnées des points
141     theta_reel, theta_0, theta_a [array (K, 3)] : angles de rotation des
142     caméras, respectivement initiaux, perturbés et réestimés
143     x_reel, x_0, x_a [array (N, K, 2)] : coordonnées des images des points,
144     respectivement initiaux, perturbés et réestimés
145
146     """
147
148     C_reel, X_reel, theta_reel = genere_donnees()
149
150     if affichage: scene_3D()
151
152     matrices_P = []
153     for i in range(K):
154         matrices_P.append(matrice_P(C_reel[i], theta_reel[i], f))
155
156     x_reel = np.zeros((N, K, 2))
157     for j in range(N):
158         for i in range(K):
159             point = matrices_P[i].dot(np.array([X_reel[j, 0], X_reel[j, 1],
160                                              X_reel[j, 2], 1]))
161             x_reel[j, i] = point[:-1] / point[-1]
162
163     # perturbation des coordonnées et des angles
164
165     x_0 = np.copy(x_reel)
166     if not x_exact: x_0 += normal(0, sigma_x, (N, K, 2))
167     theta_0 = theta_reel + normal(0, sigma_theta, (K, 3))
168
169     # minimisation de F
170
171     A = matrice_A(C_reel, theta_0, x_0, f)
172
173     # ajout du terme forcant sur la matrice M
174     contrainte = np.eye(3*K) * lamb_theta/sigma_theta
175     M = csr_matrix(np.append(A, contrainte, axis=0))

```

```
169
170     b = np.zeros((K*(K-1)//2*N+3*K))
171     b[:K*(K-1)//2*N] = -F(C_reel, theta_0, x_0, f)
172
173     erreur = ss1.lsqr(A, -F(C_reel, theta_0, x_0, f), conlim=1.0e+8)[0]
174
175     theta_a = theta_0 + erreur.reshape((K, 3))
176     x_a = np.copy(x_0)
177
178     return C_reel, X_reel, theta_reel, theta_0, theta_a, x_reel, x_0, x_a
```

Pour l'affichage des différentes figures présentes sur ce rapport, on pourra se référer directement aux fichiers BA_simulations.py et BA_simulations_terme_forçant.py sur Github.