Filtre de Kalman : versions CPU et GPU

Introduction

Ce document présente dans un premier temps les classes développées pour le filtre de Kalman et leur caractéristiques (méthodes, attributs, variables). Dans un second temps, les versions CPU et GPU sont comparées à la version MATLAB, dans différentes configurations, pour connaître leurs performances, au niveau du temps d'exécution et de la fiabilité des résultats.

Notations

D_Mo : matrice de l'ASO, permet de passer des phases aux pentes.
N_Act : matrice d'influence normalisée par *unitpervolt* (à valeurs entre 0 et *unitpervolt*), permet de passer des tensions aux phases.
PROJ : inverse de N_Act, permet de passer des phases aux tensions.
SigmaV : matrice de covariance du bruit de modèle.
bruit : valeur moyenne de la diagonale de la matrice de covariance du bruit de mesure.
SigmaTur : matrice de covariance de la turbulence.
pixsize : angle sur le ciel permettant d'obtenir un pixel. Dans le cas, où on se place à Shannon (2 pixels pour un déphasage de *lambda_ASO/2/pi*), on a *pixsize [arcsec/px] = lambda_ASO/2/d = lambda_ASO/2/(D/Nssp)*180/pi*3600* ; avec d le diamètre d'une microlentille ; D le diamètre du miroir principal ; Nssp le nombre de sous-pupilles en linéaire.
unitpervolt : coefficient (en micron/V) correspondant à la différence de marche au sommet d'un actionneur pour lequel on a appliqué 1V.

La phase résiduelle est définie comme la somme de la phase turbulente et de la phase de correction.

Partie A : Version standalone du filtre de Kalman

I) Les classes *kp_kalman_core_sparse* et *kp_kalman_core_full*

Le coeur du filtre de kalman est constitué par les classes *kp_kalman_core** (classes commençant par *kp_kalman_core)*.

Selon que les matrices *D_Mo*, *N_Act* et *PROJ* soient toutes pleines ou toutes creuses, on utilise les classes *kp_kalman_core_full** ou *kp_kalman_core_sparse**. Ces deux groupes de classes sont complètement indépendantes.

Les classes *kp_kalman_core_sparse et kp_kalman_core_full* sont deux classes abstraites, qui sont dérivées en deux classes chacune : une classe utilisée pour calculer sur CPU (*kp_kalman_core_sparse_CPU* et *kp_kalman_core_full_CPU*) et une autre classe utilisée pour calculer sur GPU (*kp_kalman_core_sparse_GPU* et *kp_kalman_core_full_GPU*).

Avec chaque type de classe (sparse ou full, CPU ou GPU), il est possible d'utiliser le filtre de Kalman :

- en base modale ou zonale,
- avec un modèle autorégressif d'ordre 1 ou d'ordre 2

Les versions CPU et GPU sont similaires au niveau algorithmique. Seule la manière d'exécuter les calculs diffère.
Pour la version CPU, les calculs matriciels sont effectués avec la bibliothèque MKL (Intel® Math Kernel Library).
Pour la version GPU, les calculs matricels sont effectués avec les bibliothèques :

- CUBLAS pour les matrices pleines,
- CUSPARSE pour les matrices creuses,
- MAGMA pour les inversions de matrices.
- THRUST pour les réductions (addition des éléments d'un vecteur ou d'une matrice)

Comme les classes pour CPU/GPU sont très proches dans leur conception (attributs, méthodes), la suite du document détaille uniquement les classes pour GPU.

1) Les principaux attributs

Attributs correspondant aux différentes dimensions :

- *nb_p* : nombre de pentes
- *nb_act* : nombre d'actionneurs
- *nb_z* : nombre de modes Zernikes sans le piston (utilisé uniquement en modal)

- *nb_az* : *nb_az=nb_act* si zonal, *nb_az=nb_z* si modal
- *nb_n* : *nb_n=2*nb_az*, nombre d'éléments du vecteur d'état

Attributs correspondant à des vecteurs ou matrices sur GPU

- *cu_Y_k* : mesure à l'instant k. Vecteur copié à partir du paramètre de la méthode next_step
- *cu_U_k*, *cu_U_km1*, *cu_U_km2* : vecteurs des tensions aux instants *k*, *k-1* et *k-2*
- *cu_Xkp1sk* et *cu_X_kskm1* : estimations *a priori* de l'état aux instants *k+1* et *k*, sachant les observations jusqu'aux instants *k* et *k-1*
- *cu_atur* : sous-matrice *A1(1:nb_n/2 , 1:nb_n/2)*. Ce vecteur est initialisé lors du premier appel de la méthode calculate_gain.
- *cu_btur* : sous-matrice *A1(1:nb_n/2 , nb_n/2+1:nb_n)*. Ce vecteur est initialisé lors du premier appel de la méthode calculate_gain.
- *cu_H_inf* : gain de Kalman, matrice initialisée à la fin du premier appel de la méthode calculate_gain
- *cu_D_Mo*, *cu_N_Act*, *cu_PROJ* : matrices initialisées lors de la création de l'objet.

Autres attributs :

- gainComputed : booleen permettant de verifier que le gain de Kalman a ete calcule. Initialise a false dans le constructeur puis fixe a true a la fin du calcul du gain
- *ordreAR* : ordre du modele auto-régressif (1 ou 2). Cette valeur est définie en fonction du vecteur *btur* : si tous ses éléments sont nuls, *ordreAR=1*, sinon *ordreAR=2*.
- isZonal : booléen correspondant à la base utilisée : zonale (true) ou modale (false)

Les « Attributs correspond aux différentes dimensions » et les « Autres attributs » sont membre de la classe mère (et des classes filles par héritage) alors que les « Attributs correspondant à des vecteurs ou matrices » sont membres des classes filles uniquement.

Il existe également d'autres attributs dans ces classes, qui sont des vecteurs utilisés uniquement comme résultats intermédiaires d'opérations de la méthode *next_step*. Il aurait été possible de déclarer ces vecteurs comme variables de la méthode *next_step*, mais cela impliquerait l'allocation et la libération de mémoire sur GPU à chaque appel de la méthode *next_step*. En étant membres de la classe, ces vecteurs sont alloués sur le GPU lors de la création de l'objet et sont libérés lors de sa destruction.

2) Le constructeur

Le constructeur de ces classes prend comme paramètres :

- la matrice *D_Mo* de dimension *(nb_p , nb_az)*,
- la matrice *N_Act* de dimension *(nb_az , nb_act)*,
- la matrice *PROJ* de dimension *(nb_act , nb_az)*,
- une chaîne de caractère valant, soit « zonal », soit « modal », en fonction de la base considérée.

Les trois matrices *D_Mo, N_Act et PROJ* doivent être toutes creuses pour *kp_kalman_core_sparse_GPU* ou toutes pleines pour *kp_kalman_core_full_GPU*.
Lors de la crétion d'un objet de ces classes, les attributs *nb_p, nb_act, nb_z, nb_az* et *nb_n* sont

initialisés en fonction des dimensions des matrices *D_Mo, N_Act et PROJ*. Les trois matrices *D_Mo*, *N_Act* et *PROJ*, sont copiées sur GPU. Toutes les variables utilisées par *next_step* sont allouées sur GPU, en tant qu'attributs de la classe (sauf *cu_H_inf* car taille importante).

3) La méthode *calculate_gain*

La méthode c*alculate_gain* permet de calculer le gain de kalman *H_inf*, avec l'équation de Riccati. Elle prend comme paramètres :

- *bruit* : valeur moyenne de la diagonale de la matrice de covariance du bruit de mesure
- *k_W* :paramètre libre, choisit par l'utilisateur pour compenser les erreurs de modèle et de mesures afin d'optimiser la performance (rapport de Strehl)
- *SigmaV* de dimension (*nb_az* , *nb_az*) : matrice de covariance du bruit de modèle
- vecteur *atur* de dimension *nb_az*, sous-matrice *A1(1:nb_n/2 , 1:nb_n/2)*
- vecteur *btur* de dimension *nb_az*, sous-matrice *A1(1:nb_n/2 , nb_n/2+1:nb_n)*

Tous les éléments non-diagonaux des sous-matrices *A1(1:nb_n/2 , 1:nb_n/2)* et *A1(1:nb_n/2 , nb_n/2+1:nb_n)* sont supposés nuls.

L'ordre du modèle autorégressif ne peut être que 1 ou 2. Il est automatiquement déterminé en fonction du vecteur *btur* (un des arguments de cette méthode). Le modèle considéré est AR1, si tous les éléments de ce vecteur sont nuls, ou AR2 sinon.

Les principales actions effectuées par cette méthode sont :

- La matrice *SigmaV* et les vecteurs *atur* et *btur* sont copiés sur GPU.
- En fonction du vecteur *btur*, on considère un modèle de type AR1 ou AR2.
- Calcul itératif des matrices *alpha_kp1*, *beta_kp1* et *T_kp1*
- Calcul du gain de Kalman *H_inf*, à partir de la dernière matrice *T_kp1*, en utilisant des matrices réduites dans le cas de l'AR1.

De manière à optimiser l'espace alloué sur GPU, *cu_H_inf* est désalloué au début de la méthode *calculate_gain* et réalloué à la fin de la méthode, lorsque le nouveau *cu_H_inf* est calculé.
Tous les calculs de cette méthode sont effectués sur GPU. A la fin de cette méthode, les vecteurs *atur*, *btur* et la matrice *H_inf* (attributs de la classe) sont toujours présents sur le GPU, car ils sont nécessaires à la méthode *next_step*.

Les objets (matrices et vecteurs) relatives à cette méthode sont et doivent être en double précision pour s'assurer de la convergence pour obtenir un gain de Kalman correct. Le paragraphe Partie A : VII) 1) détaille les conséquences dues à la précision lorsque les calculs pour le gain de Kalman sont effectués en simple précision.

4) La méthode *next_step*

La méthode *next_step* permet de calculer le vecteur des tensions *U_k* à partir des pentes *Y_k*. La méthode *calculate_gain* doit être appelée avant le premier appel à la méthode *next_step*, car cette méthode nécessite que le gain de Kalman *H_inf* soit déja calculé (par *calculate_gain*).

Elle prend comme paramètres :

- vecteur *Y_k*, de dimension *nb_p*, contenant les pentes.
- vecteur *U_k*, de dimension *nb_act*, contenant les tensions, qui sera mis à jour à la fin de la méthode.

Les valeurs des matrices et vecteurs de la méthode next_step sont toutes, soit en simple, soit en double precision. Par défaut, les valeurs sont en simple précision, mais si la variable de compilation *KP_DOUBLE* est définie lors de la compilation alors ces objets seront en double précision.
Cette variable de compilation est définie (via le Makefile) si la variable d'environnement *COMPILATION_LAM* vaut « double ».

II) Les classes templates pour vecteurs et matrices

Des classes templates *kp_cu_vector*, *kp_cu_matrix*, *kp_cu_smatrix* ont été créées de manière à pouvoir être utilisées facilement avec les classes *kp_vector*, *kp_matrix*, *kp_smatrix*. Par exemple si, *mat* est une instance de la classe template *kp_matrix<float>*ou *kp_matrix<double>*, alors il est possible de copier les données de cette matrice sur GPU en écrivant simplement : *kp_cu_matrix<double> cu_mat(mat);* . Lors de la création de l'objet *cu_mat*, les données de la matrice *mat* sont copiées sur GPU, et toutes ses caractéristiques (nombre de lignes, de colonnes, etc.) sont copiées dans des attributs de l'objet, sur CPU.

De manière générale, les classes, objets, attributs et variables contenant «cu_ » correspondent à des données sur GPU.

Avec ces ces classes, il est possible de créer une matrice contenant des données en double précision à partir d'une matrice en simple précision (et vice versa). Si mat est une instance de kp_matrix<float>, il suffit d'écrire *kp_matrix<double> mat_double(mat);* pour créer une matrice *mat_double*, contenant les mêmes données que *mat*, mais en double précision.

Il est aussi possible de créer, directement par un contructeur, une matrice sur GPU (aussi bien en simple qu'en double précision) à partir d'une matrice sur CPU ou GPU, en simple ou double precision.

Il est également possible de créer par un contructeur, une matrice sur CPU (aussi bien en simple qu'en double précision) à partir d'une matrice sur CPU en simple ou double precision.

Par contre, pour copier une matrice sur CPU à partir d'une matrice sur GPU, il faut d'abord crée un objet de classe *kp_matrix<float/double>* puis passer par la fonction de prototype *void kp_cu2kp_matrix(kp_matrix, kp_cu_matrix)*, où le premier argument de classe *kp_matrix* est la matrice de destination sur CPU et *kp_cu_matrix* est la matrice source sur GPU.

Ces possibilitées de copies entre matrices sont également valables pour des copies entre vecteurs avec les classes *kp_vector*, *kp_cu_vector* et la fonction *kp_cu2kp_vecto*r pour les copies GPU vers CPU. C'est également le cas pour les matrices creuses avec les classes *kp_smatrix*, *kp_cu_smatrix* et la fonction *kp_cu2kp_smatrix* pour les copies GPU vers CPU.

En plus du contructeur, l'opérateur = a été redéfini pour ces classes. Il est donc possible de copier une matrice (respectivement un vecteur, respectivement une matrice creuse) sur GPU en simple ou double précision, à partir d'une matrice (respectivement d'un vecteur, respectivement d'une matrice creuse) sur CPU ou GPU en simple ou double précision.

III) Compilation

Pour compiler la version standalone de Kalman, il faut d'abord compiler les classes kp_kalman_core_* et les classes utilisées par celles-ci en exécutant le Makefile dans lam/. Ce Makefile va générer deux bibliothèques statiques : lib_kalman.a et lib_kp_classes.a.
Il faut ensuite exécuter le Makefile dans lam/kalman_CPU_GPU/test, qui va génerer les executables kalman_CPU et kalman_GPU. Pour exécuter ce deuxième Makefile, la bibliothèque matio est nécessaire.

IV) Ecriture de la boucle d'OA

Pour calculer les tensions U_k à partir des pentes Y_k, deux écritures ont été testées :

- celle de l'article de Cyril Petit : « LQG control for adaptive optics and multiconjugate adaptive optics : experimental and numerical analysis » (JOSAA 2009), faisant intervenir la matrice *H_inf*.
- celle du du document de Cyril, faisant intervenir la matrice *L_inf*.

1) Méthode zonale

Dans le cas de la méthode zonale, l'écriture de l'article (*H_inf*) donne de meilleures performances (en temps de calcul) que celle du document (*L_inf*). Cela est vrai pour la version CPU, comme pour la version GPU.
Le tableau suivant compare les temps d'exécution des deux versions, dans le cas du 40m, en double precision.

zonal, 40m double precision	Temps L_inf (s)		Temps H_inf (s)		Temps L_inf / Temps H_inf	
	sparse	full	sparse	full	sparse	full
CPU	235	311	145	242	1,62	1,29
GPU	53,6	75,2	36,7	66,4	1,46	1,13

2) Méthode modale

Dans le cas de la méthode modale, seule la simulation sur 8m a été effectuée, car au-delà de 8m, le nombre de modes de Zernike nécessaire à la configuration d'OA choisit dans nos tests est trop élevé (impossibilité numérique de générer suffisamment de Zernike). La méthode utilisant les Karhuen-Loeve n'a pas été testé, car les matrices ne sont pas disponibles pour le moments.
Le tableau ci-dessous n'est probablement pas représentatif de ce qu'on aurait pour des plus gros diamètre, mais les écritures se valent. Le temps de calcul en matrice creuse (sparse) n'est pas réellement à prendre en compte, car toutes les matrices sont pleines (on a convertit des matrices pleines sous format de matrices creuses).
On constate que sur GPU, les deux écritures se valent. Sur CPU, l'écriture de l'article est la meilleure.

modal, 8m double precision	Temps L_inf (s)		Temps H_inf (s)		Temps L_inf / Temps H_inf	
	sparse	full	sparse	full	sparse	full
CPU	2,84	1,05	3,05	0,72	0,93	1,46
GPU	1,52	1,35	1,5	1,37	1,01	0,99

Comme l'écriture figurant dans l'article semble meilleure que celle dans le document, tous les tests

de ce document ont été effectués avec l'écriture de l'article.

V) Stockage en mémoire

Toutes les matrices pleines, qu'elles soient définies sur CPU ou sur GPU sont stockées en mémoire comme une succession de vecteurs colonnes (format column-major).

Pour les matrices creuses, il est possible de les stocker, soit en column-major, soit en row-major.

1) Formats row-major et column-major pour une matrice creuse

Pour représenter une matrice creuse, trois tableaux, sont utilisés : rowind (indices de lignes), colind (indices des colonnes), values (valeurs). Chacun de ces tableaux est de taille NNZ (nombre d'éléments non nuls de la matrice).

Si on prend la matrice A suivante :

```
A =

     0    77     0    80     0
    96    82    40    43    26
     0    87    26    91     0
```

Le format column-major est utilisé lorsque les éléments non-nuls sont stockés d'abord par colonne, puis par ligne. Dans cet exemple :

```
A_col_maj =

   (2,1)       96
   (1,2)       77
   (2,2)       82
   (3,2)       87
   (2,3)       40
   (3,3)       26
   (1,4)       80
   (2,4)       43
   (3,4)       91
   (2,5)       26
```

Le format row-major est utilisé lorsque les éléments non-nuls sont stockés d'abord par ligne, puis par colonne. Dans cet exemple :

```
A_row_maj =

   (1,2)       77
   (1,4)       80
   (2,1)       96
   (2,2)       82
   (2,3)       40
   (2,4)       43
   (2,5)       26
   (3,2)       87
   (3,3)       26
   (3,4)       91
```

2) Comparaison des performances des multiplications d'une matrice creuse entre row-major et column-major

Les tableaux suivants montrent l'influence du stockage des matrices creuses entre row-major et column-major.
Pour faire une multiplication entre une matrice creuse A et une matrice pleine ou un vecteur B, il est plus intéressant (au niveau du temps d'exécution) que la matrice creuse A soit en row-major. Par contre, si on veut calculer la multiplication entre la transposée de A et B, il est préférable que A soit en column-major.

Temps de calcul (en secondes) pour 5000 multiplications successives entre la matrice carée creuse N_Act (ou de sa transposée) par un vecteur (cas zonal, AR1,double précision)

		8m	16m	32m	40m
	nbre de lignes	5 209	20 365	80 653	125 825
	NNZ	241	877	3349	5185
Matrice creuse N_Act en column-major	pas de transposition	0,384	2,04	2,23	2,8
	transposition	0,111	0,113	0,154	0,188
Matrice creuse N_Act en row-major	pas de transposition	0,111	0,112	0,154	0,188
	transposition	0,379	1,97	2,25	2,84

Temps de calcul (en secondes) pour 50 multiplications successives entre la matrice carée creuse N_Act (ou de sa transposée) par une matrice carée de même taille que N_Act (cas zonal, AR1,double précision)

		8m	16m	32m	40m
	nbre de lignes	5 209	20 365	80 653	125 825
	NNZ	241	877	3349	5185
Matrice creuse N_Act en column-major	pas de transposition	0,0545	0,418	8,59	22
	transposition	0,0065	0,0535	0,786	1,96
Matrice creuse N_Act en row-major	pas de transposition	0,00652	0,0536	0,787	1,96
	transposition	0,0545	0,418	8,59	22

3) Taille mémoire sur GPU

L'espace mémoire alloué sur GPU pour un vecteur de classe cu_vector de N éléments est composé d'un seul tableau contenant le nombre d'élément du vecteur. Sa taille en mémoire est donc ***N*sizeof(KFPP)***, avec KFPP=float ou KFPP=double selon le type choisit lors de la compilation,

L'espace mémoire alloué sur GPU pour un matrice pleine de classe cu_matrix de M lignes et N colonnes est composé d'un seul tableau contenant le nombre d'élément de la matrice. Sa taille en mémoire est donc ***M*N*sizeof(KFPP)***, avec *KFPP=float* ou *KFPP=double* selon le type choisit lors de la compilation.

L'espace mémoire alloué sur GPU pour une matrice creuse de classe cu_smatrix M lignes, N colonnes et NNZ éléments non nuls est composé de 5 tableaux :

- *rowind_cu* : tableau constitué de NNZ entiers, correspondant aux indices de ligne des éléments non nuls de la matrice
- *colind_cu* : tableau constitué de NNZ entiers, correspondant aux indices de colonne des éléments non nuls de la matrice
- *values_cu* : tableau constitué de NNZ KFPP (float ou double), correspondant aux valeurs des éléments non nuls de la matrice
- *csrRowPtr* : tableau constitué de M+1 entiers ; les M premières valeurs correspondent aux indices (allant de 0 à NNZ-1) du premier élément non nul de chaque ligne, la dernière valeur correspond à *NNZ+csrRowPtr(0)*.
- *csrRowPtrT* : idem que *csrRowPtr*, mais pour la matrice transposée.

La taille totale en mémoire est donc de *(2*NNZ+M+N+2)*sizeof(int) + NNZ*sizeof(KFPP)*.
Dans le cas d'une matrice creuse en double précision (*sizeof(KFPP) = 8 octets ; sizeof(int) = 4 octets*), la taille mémoire de la matrice creuse sera plus faible qu'en matrice pleine si :

$$NNZ < 1/4\,(2MN - M - N - 2)$$

ou $NNZ < 1/2\,(MN)$ *si* $M>>1$ *et* $N>>1$

La mémoire allouée sur GPU est la plus importante lors du calcul récursif des matrices *alpha_kp1*, *beta_kp1* et *T_kp1*, nécessaires pour calculer le gain de Kalman *H_inf*.
Dans le cas en base zonale, modèle AR1, double précision pour 40m, l'espace maximal alloué sur GPU est de :

- 2,6 Go si *D_Mo, N_Act* et *PROJ* sont considérées comme matrices creuses,
- 3,8 Go si *D_Mo, N_Act* et *PROJ* sont considérées comme matrices pleines.

Sur les figures suivantes, on représente les variables allouées (en vert) et non allouées (en gris) sur GPU dans le temps (différentes colonnes).

Ce tableau montre comment la mémoire est utilisée sur GPU dans le cas des matrices pleines.

	Cas AR1, matrices toutes pleines, Zonal, 40m, Double précision	Dimensions	Nbre d'éléments	Taille Mémoire (ko)	construction de l'objet	initialisation Riccati							boucle Riccati	fin boucle Riccati	Calcul gain Kalman								fin Riccati	boucle kalman
attributs de la classe	cu_U_km2	NB_ACT	5 185	41																				
	cu_U_km1	NB_ACT	5 185	41																				
	cu_X_kskm1	NB_N	10 370	83																				
	cu_D_Mo	NB_P x NB_AZ	52 098 880	416 791																				
	cu_N_Act	NB_AZ x NB_ACT	26 884 225	215 074																				
	cu_PROJ	NB_ACT x NB_AZ	26 884 225	215 074																				
	cu_Y_k	NB_P	10 048	80																				
	cu_Nact_Ukm2	NB_AZ	5 185	41																				
	cu_tmp_vec1	NB_AZ	5 185	41																				
	cu_innovation	NB_P	10 048	80																				
	cu_X_kskm1_tmp	NB_N	10 370	83																				
	cu_X_kp1sk	NB_N	10 370	83																				
	cu_X_kp1sk_tmp	NB_AZ	5 185	41																				
	cu_Y_kskm1	NB_P	10 048	80																				
	cu_A1_00_Xkdebut	NB_AZ	5 185	41																				
	cu_A1_01_Xkfin	NB_AZ	5 185	41																				
	cu_X_kp1sk_debut	NB_AZ	5 185	41																				
	cu_U_k	NB_ACT	5 185	41																				
	cu_H_inf	NB_N x NB_P	104 197 760	833 582																				
	cu_atur	NB_AZ	5 185	41																				
	cu_btur	NB_AZ	5 185	41																				
variables de Riccati	cu_alpha_kp1	NB_AZ x NB_AZ	26 884 225	215 074																				
	cu_beta_kp1	NB_AZ x NB_AZ	26 884 225	215 074																				
	cu_zeros_nbp_nbaz	NB_P x NB_AZ	52 098 880	416 791																				
	cu_zeros_Dmo	NB_P x NB_N	104 197 760	833 582																				
	cu_C1	NB_P x NB_AZ	52 098 880	416 791																				
	cu_SigmaV	NB_AZ x NB_AZ	26 884 225	215 074																				
	cu_T_kp1sk	NB_AZ x NB_AZ	26 884 225	215 074																				
	cu_alpha_k	NB_AZ x NB_AZ	26 884 225	215 074																				
	cu_T_k	NB_AZ x NB_AZ	26 884 225	215 074																				
	cu_beta_k	NB_AZ x NB_AZ	26 884 225	215 074																				
	cu_lBG_1	NB_AZ x NB_AZ	26 884 225	215 074																				
	cu_Tk_lBG1	NB_AZ x NB_AZ	26 884 225	215 074																				
	cu_betak_Tk	NB_AZ x NB_AZ	26 884 225	215 074																				
	cu_alphak_lBG1	NB_AZ x NB_AZ	26 884 225	215 074																				
	cu_alphak_lBG1_betak	NB_AZ x NB_AZ	26 884 225	215 074																				
	cu_Tk_lBG1_alphak	NB_AZ x NB_AZ	26 884 225	215 074																				
	cu_diag_cu_Tkp1sk	NB_AZ	5 185	41																				
	cu_diag_cu_Tk	NB_AZ	5 185	41																				
	cu_Tkp1sk_C1t	NB_AZ x NB_P	52 098 880	416 791																				
	cu_H_opt	NB_AZ x NB_P	52 098 880	416 791																				
	cu_Atur_Hopt	NB_AZ x NB_P	52 098 880	416 791																				
	cu_Sigma_tot	NB_P x NB_P	100 962 304	807 698																				
	Taille totale sur GPU (Go)				**0,8**	**2,5**	**2,1**	**2,5**	**1,7**	**1,9**	**1,7**	**3,8**	**3,8**	**1,5**	**1,9**	**1,7**	**2,5**	**2,9**	**1,7**	**2,1**	**2,9**	**2,1**	**1,7**	**1,7**

Ce tableau montre comment la mémoire est utilisée sur GPU dans le cas des matrices creuses.

	Cas AR1, Avec matrices creuses, Zonal, 40m, Double précision, Gain expression 2	**Dimensions**	**Nbre d'éléments (non nuls pour matrices creuses)**	**Taille Mémoire (ko)**	**construction de l'objet**	**initialisation Riccati**					**boucle Riccati**	**fin boucle Riccati**	**calcul gain kalman**												**fin Riccati**	**init4kalman**	**boucle kalman**
attributs de la classe	cu_U_km2	NB_ACT	5 185	41																							
	cu_U_km1	NB_ACT	5 185	41																							
	cu_X_kskm1	NB_N	10 370	83																							
	cu_D_Mo	NB_P x NB_AZ	40 192	704																							
	cu_N_Act	NB_AZ x NB_ACT	125 825	2 055																							
	cu_PROJ	NB_ACT x NB_AZ	3 278 054	52 490																							
	cu_Y_k	NB_P	10 048	80																							
	cu_Nact_Ukm2	NB_AZ	5 185	41																							
	cu_tmp_vec1	NB_AZ	5 185	41																							
	cu_innovation	NB_P	10 048	80																							
	cu_X_kskm1_tmp	NB_N	10 370	83																							
	cu_X_kp1sk	NB_N	10 370	83																							
	cu_X_kp1sk_tmp	NB_AZ	5 185	41																							
	cu_Y_kskm1	NB_P	10 048	80																							
	cu_A1_00_Xkdebut	NB_AZ	5 185	41																							
	cu_A1_01_Xkfin	NB_AZ	5 185	41																							
	cu_X_kp1sk_debut	NB_AZ	5 185	41																							
	cu_U_k	NB_ACT	5 185	41																							
	cu_H_inf	NB_N x NB_P	104 197 760	833 582																							
	cu_atur	NB_AZ	5 185	41																							
	cu_btur	NB_AZ	5 185	41																							
variables de Riccati	cu_alpha_kp1	NB_AZ x NB_AZ	26 884 225	215 074																							
	cu_beta_kp1	NB_AZ x NB_AZ	26 884 225	215 074																							
	cu_C1	NB_P x NB_AZ	40 192	322																							
	cu_C1_dense	NB_P x NB_AZ	52 098 880	416 791																							
	cu_SigmaV	NB_AZ x NB_AZ	26 884 225	215 074																							
	cu_T_kp1sk	NB_AZ x NB_AZ	26 884 225	215 074																							
	cu_alpha_k	NB_AZ x NB_AZ	26 884 225	215 074																							
	cu_T_k	NB_AZ x NB_AZ	26 884 225	215 074																							
	cu_beta_k	NB_AZ x NB_AZ	26 884 225	215 074																							
	cu_IBG_1	NB_AZ x NB_AZ	26 884 225	215 074																							
	cu_Tk_IBG1	NB_AZ x NB_AZ	26 884 225	215 074																							
	cu_betak_Tk	NB_AZ x NB_AZ	26 884 225	215 074																							
	cu_alphak_IBG1	NB_AZ x NB_AZ	26 884 225	215 074																							
	cu_alphak_IBG1_betak	NB_AZ x NB_AZ	26 884 225	215 074																							
	cu_Tk_IBG1_alphak	NB_AZ x NB_AZ	26 884 225	215 074																							
	cu_diag_cu_Tkp1sk	NB_AZ	5 185	41																							
	cu_diag_cu_Tk	NB_AZ	5 185	41																							
	cu_Tkp1skt	NB_AZ x NB_AZ	26 884 225	215 074																							
	cu_Tkp1sk_C1t_tmp	NB_P x NB_AZ	52 098 880	416 791																							
	cu_Tkp1sk_C1t	NB_AZ x NB_P	52 098 880	416 791																							
	cu_H_opt	NB_AZ x NB_P	52 098 880	416 791																							
	cu_Atur_Hopt	NB_AZ x NB_P	52 098 880	416 791																							
	cu_Sigma_tot	NB_P x NB_P	100 962 304	807 698																							
	Taille totale sur GPU (Go)				**0,1**	**0,9**	**0,5**	**0,9**	**0,7**	**2,6**	**2,6**	**0,3**	**0,3**	**0,5**	**0,7**	**0,5**	**0,9**	**0,5**	**1,3**	**1,7**	**0,5**	**0,9**	**1,7**	**0,9**	**0,9**	**0,9**	**0,9**

VI) Comparaison des performances temporelles entre les versions CPU et GPU

La comparaison des performances en temps d'exécution entre les versions CPU et GPU a été éffectuée avec

- 2 processeurs Intel Xeon CPU E5-2680 v2 @ 2.80GHz, de 10 cores et 25 Mo de cache chacun, pour la version CPU
- GPU Nvidia Tesla K20, de 13 Streaming Multiprocesseurs et 1.3Mo de cache L2, pour la version GPU

Dans la suite du document, pour comparer les performances (en temps de calcul) des versions CPU et GPU, on parlera de gain, définit comme le rapport du temps d'exécution en CPU, sur le temps d'exécution en GPU

1) Calcul du gain de Kalman *H_inf* par l'équation de Riccati, matrices creuses stockées en column-major

Il y a un gain d'environ 2 pour les grands diamètres (32m et 40m). Ce gain de temps est limité par l'inversion, car c'est l'opération qui est la plus coûteuse et sur laquelle le gain est assez faible (environ 1,15). Ce gain est un peu plus important lorsque la matrice D_Mo est considérée pleine plutôt que creuse.

On remarque que les temps de calculs entre la matrice D_Mo creuse (sparse) et la matricee D_Mo pleine (full) sont très proches pour les calculs de *alpha_k*, *beta_k*, *T_k* et de l'inversion, car dans la boucle faisant intervenir ces 4 calculs, la matrice D_Mo n'intervient pas. Elle intervient seulement à l'initialisation (pour calculer *beta_0*, et à la fin, pour calculer *H_inf* à partir de *T_k*.

Dans le cas du 8m, on observe que la version CPU est plus rapide que la version GPU pour les matrices creuses. Cela peut être expliqué par le fait que dans le cas d'une multiplication d'une matrice creuse sur GPU, il est nécessaire de convertir la matrice du format COO en CSR avant de réaliser la multiplication proprement dite.

Temps de calcul de la matrice H_inf par Riccati

zonal, AR1, column-major Double precision			temps GPU (s) (11 itérations)	temps CPU (s) (11 itérations)	temps CPU / temps GPU
sparse	8m	alpha_k	0,00188	0,00412	2,19
		beta_k	0,00191	0,00372	1,95
		T_k	0,00299	0,0058	1,94
		inversion	0,0832	0,0188	0,23
		Total 1	**0,12**	**0,059**	**0,49**
	16m	alpha_k	0,0352	0,0979	2,78
		beta_k	0,034	0,0948	2,79
		T_k	0,0518	0,142	2,74
		inversion	0,313	0,25	0,80
		Total 1	**0,577**	**0,893**	**1,55**
	32m	alpha_k	1,64	4,34	2,65
		beta_k	1,6	4,28	2,68
		T_k	2,53	6,47	2,56
		inversion	3,99	4,56	1,14
		Total 1	**13,4**	**27,7**	**2,07**
	40m	alpha_k	6,02	15,4	2,56
		beta_k	5,86	15,1	2,58
		T_k	9,22	22,5	2,44
		inversion	12,7	14,5	1,14
		Total 1	**46**	**91,8**	**2,00**
full	8m	alpha_k	0,00189	0,0103	5,45
		beta_k	0,00193	0,00374	1,94
		T_k	0,00292	0,00557	1,91
		inversion	0,0892	0,0196	0,22
		Total 1	**0,119**	**0,0934**	**0,78**
	16m	alpha_k	0,0347	0,0981	2,83
		beta_k	0,0339	0,0949	2,80
		T_k	0,0525	0,143	2,72
		inversion	0,317	0,253	0,80
		Total 1	**0,578**	**1,09**	**1,89**
	32m	alpha_k	1,642	4,46	2,72
		beta_k	1,601	4,32	2,70
		T_k	2,532	6,47	2,56
		inversion	3,9	4,55	1,17
		Total 1	**13,5**	**31,3**	**2,32**
	40m	alpha_k	6,02	15,2	2,52
		beta_k	5,86	15	2,56
		T_k	9,22	22,7	2,46
		inversion	12,7	14,6	1,15
		Total 1	**47,3**	**105**	**2,22**

alpha_k : alpha_k*IBG_1*alpha_k;
beta_k : beta_k + alpha_k*IBG_1*beta_k*alpha_k';
T_k : T_k + alpha_k'*T_k*IBG_1*alpha_k;
Total1 correspond au temps total pour calculer H_inf à partir des différentes matrices

2) C*alc*ul des tensions *U_k* à partir des pentes *Y_k*, *matrices stockées en column-major*

Le gain entre les version CPU et GPU augmente avec le diamètre, jusqu'à un certain point (autour de 32m), avant de diminuer (matrices *D_Mo, N_Act* et *PROJ* creuses) ou de stagner (matrices pleines).

Pour des diamètres assez importants, la version GPU est meilleure que la version CPU d'un facteur 3 environ pour les matrices creuses et 3,65 pour les matrices pleines.

Temps de calcul des tension U_k à partir des pentes Y_k

zonal, AR1, column-major Double precision			temps GPU (s) (5000 itérations)	temps CPU (s) (5000 itérations)	temps CPU / temps GPU
sparse	8m	op1	0,921	0,0504	0,05
		op2	0,744	0,759	1,02
		op3	0,977	0,302	0,31
		Total 2	**2,98**	**1,13**	**0,38**
	16m	op1	1,47	0,235	0,16
		op2	1,66	3,15	1,90
		op3	2,51	2,57	1,02
		Total 2	**6,25**	**6**	**0,96**
	32m	op1	1,75	0,85	0,49
		op2	13,8	61,1	4,43
		op3	5,72	13,1	2,29
		Total 2	**23,2**	**75,2**	**3,24**
	40m	op1	4,88	1,42	0,29
		op2	32,3	119	3,68
		op3	9,56	23,9	2,50
		Total 2	**48,8**	**145**	**2,97**
full	8m	op1	0,285	0,535	1,88
		op2	0,73	0,209	0,29
		op3	0,131	0,124	0,95
		Total 2	**1,59**	**0,89**	**0,56**
	16m	op1	1,028	2,73	2,66
		op2	1,586	2,93	1,85
		op3	0,39	0,879	2,25
		Total 2	**3,68**	**6,58**	**1,79**
	32m	op1	10	45,3	4,53
		op2	13,8	46,1	3,34
		op3	3,56	16,4	4,61
		Total 2	**29,6**	**108**	**3,65**
	40m	op1	23,2	97,3	4,19
		op2	32,2	109	3,39
		op3	8,54	33,4	3,91
		Total 2	**66,38**	**242**	**3,65**

op1 : Y_kskm1 = D_Mo*(X_kskm1(nb_az+1:end) – N_Act*U_km2);
op2 : A1*(X_kskm1 + H_inf*(Y_k – Y_kskm1));
op3 : U_k = PROJ*(X_kp1sk(1:nb_az)-mean(X_kp1sk(1:nb_az))*ones(nb_az,1));
Total2 correspond au temps total pour calculer U_k à partir de Y_k

3) Influence dustockage des matrices creuses : column-major ou row-major

Le stockage en row-major des matrices *D_Mo, N_Act et PROJ* donne de meilleures performances qu'en column-major pour le calcul des *U_k* à partir des *Y_k* (rapport de temps de 1.33 pour 32m et 40m). En revanche, pour le calcul du gain de Kalman *H_inf*, les performances sont très proches.

Lorsque les matrices creuses *D_Mo*, *N_Act* et *PROJ* sont stockées au format row-major, on obtient un gain, sur la multiplication entre *PROJ* et un vecteur (op3), de 5,2 pour 40m, par rapport au column-major. Ce gain n'est pas significatif du gain gain final, car c'est l'opération *H_inf*(Y_k-Y_kskm1)* (op1) qui prend le plus de temps (très majoritairement). Comme *H_inf* est une matrice pleine, cela ne change rien, pour ce calcul que les matrices creuses soient définies en row-major ou column-major.

On remarque que pour la version CPU, contrairement à la version GPU, il est plus intéressant de stocker les matrices creuses en column-major.

Il semble assez intuitif que le gain le plus pertinent est celui obtenu en comparant la version CPU la plus rapide (matrices creuses en column-major) avec la version GPU la plus rapide (matrices creuses en row-major). Ce gain est d'environ 4 pour le 40m et monte jusqu'à 4,3 pour le 32m.

Comme le gain de Kalman H_inf, est calculé « hors ligne », c'est surtout le gain (en temps) pour obtenir les tensions à partir des pentes, qu'il faut prendre en compte.

Temps de calcul des tensions U_k à partir des pentes Y_k

zonal, AR1, Matrices Creuses, Double précision		Matrices creuses en column-major			Matrices creuses en row-major				
		temps GPU (s) (5000 itérations)	temps CPU (s) (5000 itérations)	temps CPU / temps GPU	temps GPU (s) (5000 itérations)	temps CPU (s) (5000 itérations)	temps CPU / temps GPU	temps GPU col-major / temps GPU row-major	Temps CPU col-major / temps GPU row-major
8m	op1	0,921	0,0504	0,05	0,306	0,0748	0,24	3,01	0,16
	op2	0,744	0,759	1,02	0,769	0,458	0,60	0,97	0,99
	op3	0,977	0,302	0,31	0,365	0,706	1,93	2,68	0,83
	Total 2	**2,98**	**1,13**	**0,38**	**1,75**	**1,26**	**0,72**	**1,70**	**0,65**
16m	op1	1,47	0,235	0,16	0,486	0,308	0,63	3,02	0,48
	op2	1,66	3,15	1,90	1,65	3,97	2,41	1,01	1,91
	op3	2,51	2,57	1,02	0,516	5,76	11,16	4,86	4,98
	Total 2	**6,25**	**6**	**0,96**	**3,2**	**10,1**	**3,16**	**1,95**	**1,88**
32m	op1	1,75	0,85	0,49	0,578	1,26	2,18	3,03	1,47
	op2	13,8	61,1	4,43	13,8	66,2	4,80	1,00	4,43
	op3	5,72	13,1	2,29	1,25	30,6	24,48	4,58	10,48
	Total 2	**23,2**	**75,2**	**3,24**	**17,5**	**98,2**	**5,61**	**1,33**	**4,30**
40m	op1	4,88	1,42	0,29	0,626	1,84	2,94	7,80	2,27
	op2	32,3	119	3,68	32,3	122	3,78	1,00	3,68
	op3	9,56	23,9	2,50	1,85	50,1	27,08	5,17	12,92
	Total 2	**48,8**	**145**	**2,97**	**36,7**	**174**	**4,74**	**1,33**	**3,95**

op1 : Y_kskm1 = D_Mo*(X_kskm1(nb_az+1:end) – N_Act*U_km2);
op2 : A1*(X_kskm1 + H_inf*(Y_k – Y_kskm1));
op3 : U_k = PROJ*(X_kp1sk(1:nb_az)-mean(X_kp1sk(1:nb_az))*ones(nb_az,1));
Total2 correspond au temps total pour calculer U_k à partir de Y_k

VII) Calcul en simple ou double précision

On s'intéresse ici à l'influence de la précision (simple ou double) sur les résultats et les temps de calculs des matrices et vecteurs obtenus.

Dans la suite du document, lorsqu'on parle d'erreur, on fait référence :

- pour *H_inf* : à l'erreur relative entre chaque élément de la matrice *H_inf* obtenue par rapport à la matrice H_inf_{MATLAB} de référence, obtenue par MATLAB en double précision
- pour U_k : à l'erreur relative entre la matrice *Uk* composée de 5000 lignes, correspondant à 5000 itérations, où chaque ligne correspond au vecteur *U_k* de l'itération. Le calcul de *U_k* fait intervenir un vecteur aléatoire, qui a été généré aléatoirement, au préalable, pour chaque itération puis enregistré afin d'avoir des valeurs théoriquement identiques.

1) Comparaison des résultats entre simple et double précision pour la matrice *H_inf*

Pour le calcul en double précisionde la matrice *H_inf*, les résultats obtenus correspondent à ceux obtenus avec MATLAB : une erreur centrée entre 10^{-9} et 10^{-10}, pour la version CPU comme GPU (voir figures de répartition des erreurs).

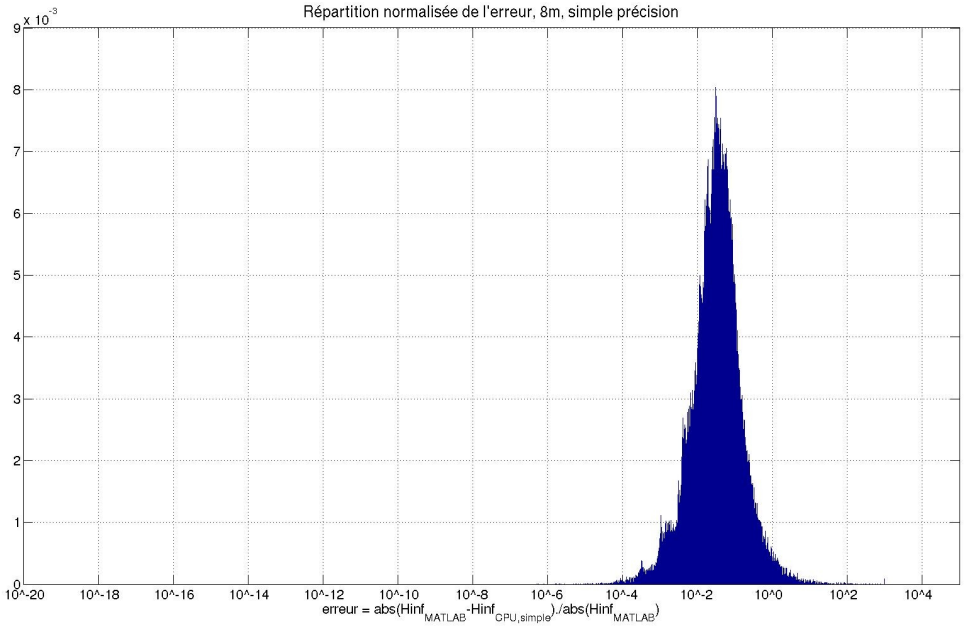
On s'aperçoit que lorsque le calcul de *H_inf* par l'équation de Riccati est effectué en simple précision, la matrice *H_inf* obtenue est bien différente de celle obtenue en double précision, que ce soit avec la version CPU ou GPU. En effet, dans le cas du 40m, presque 90 % des éléments de la matrice *H_inf* sont différents de plus de 1 % par rapport à ce qu'ils devraient être pour la version GPU, et plus de 60 % pour la version CPU.
Il est possible qu'une seule opération dans le calcul de *H_inf* soit responsable de la propagation des erreurs conduisant à une telle différence.

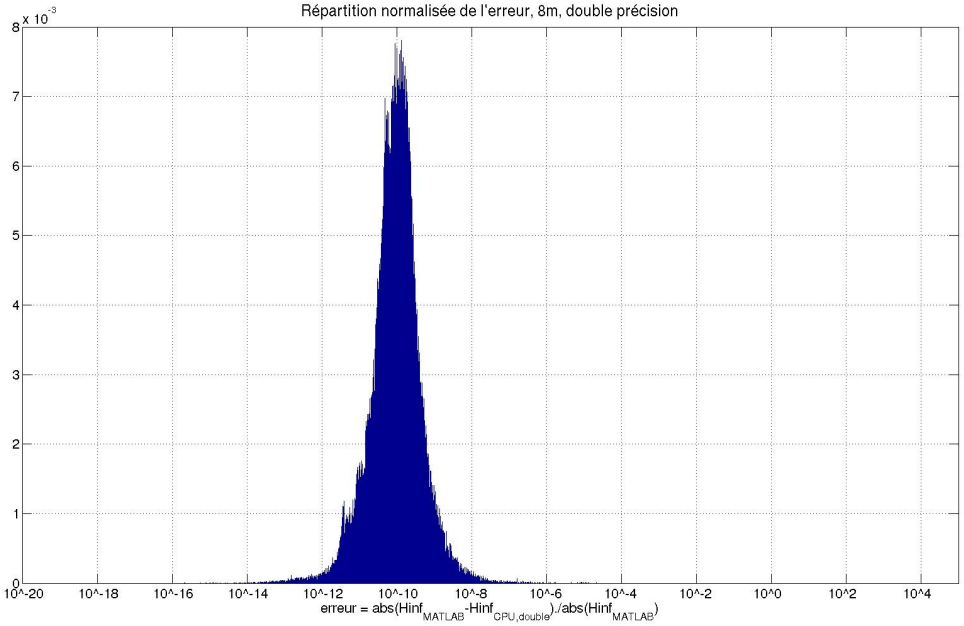
		GPU				CPU			
		8m	16m	32m	40m	8m	16m	32m	40m
H_inf en simple Précision	erreur relative max	8,15E+3	6,62E+4	2,72E+5	3,36E+6	7,29E+3	7,99E+4	3,75E+5	6,99E+5
	proportions d'éléments dont erreur < 100%	98,24%	95,34%	95,48%	76,66%	98,08%	94,97%	95,76%	96,17%
	proportions d'éléments dont erreur < 10%	81,08%	54,65%	55,66%	19,13%	80,41%	50,04%	57,84%	60,76%
	proportions d'éléments dont erreur < 5%	63,03%	34,14%	35,19%	10,28%	62,18%	30,06%	37,26%	39,90%
	proportions d'éléments dont erreur < 1%	18,86%	8,25%	8,46%	2,17%	19,23%	6,46%	9,53%	10,33%
	proportions d'éléments dont erreur < 0,1%	2,20%	0,83%	0,78%	0,22%	2,10%	0,76%	1,04%	1,04%
	proportions d'éléments dont erreur < 0,01%	0,19%	0,08%	0,07%	0,02%	0,18%	0,13%	0,13%	0,10%

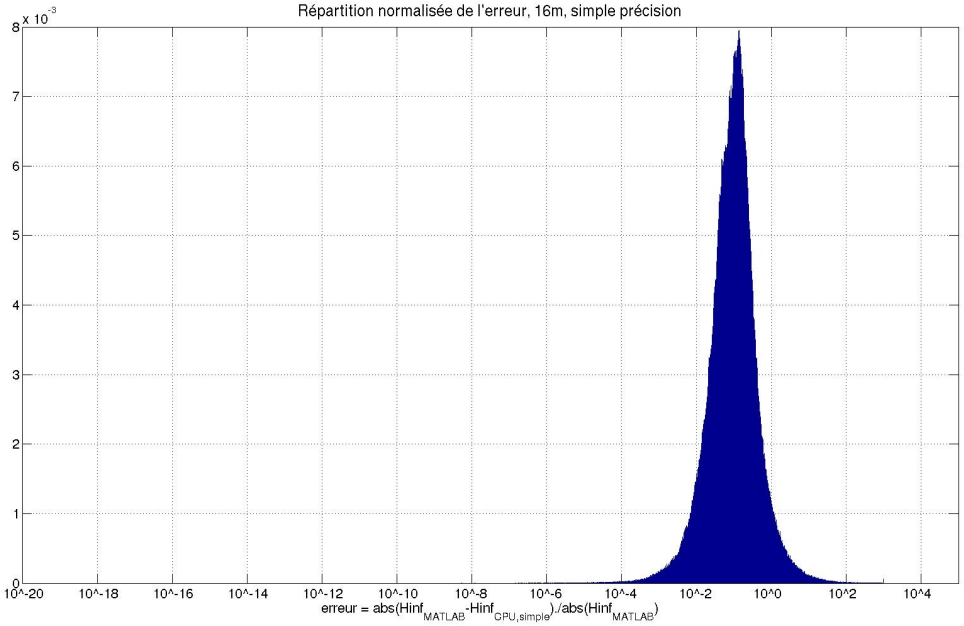
a) Répartition des erreurs relatives sur *H_inf* avec version CPU

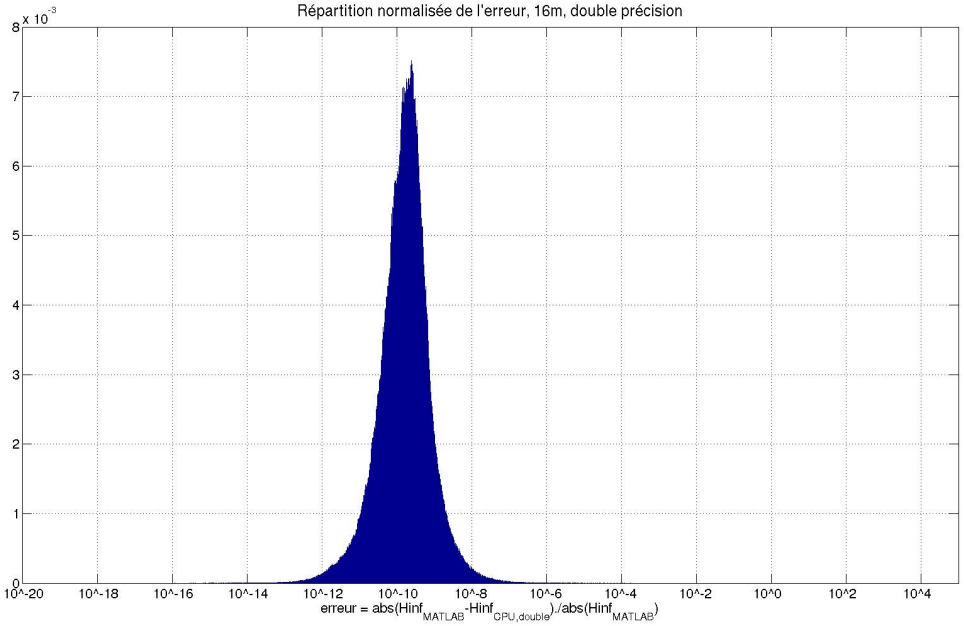
Les figures suivantes montrent de quelle manières sont réparties les erreurs relatives pour les éléments de la matrice *H_inf*, calculée par la version CPU, soit en simple précision, soit en double, pour différents diamètres.

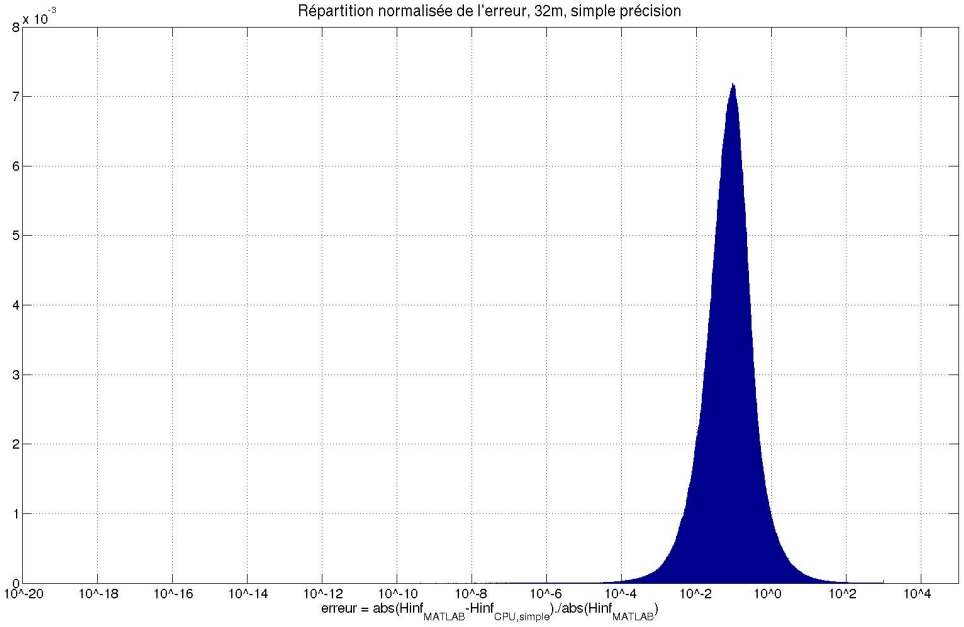
La répartition est normalisée, dans le sens où, pour chaque intervalle d'erreur (en abscisse), le nombre d'occurences correspondantes a été divisé par le nombre total d'éléments. C'est ce rapport *nombre d'occurences / nombre total*, qui est représenté en ordonnée.

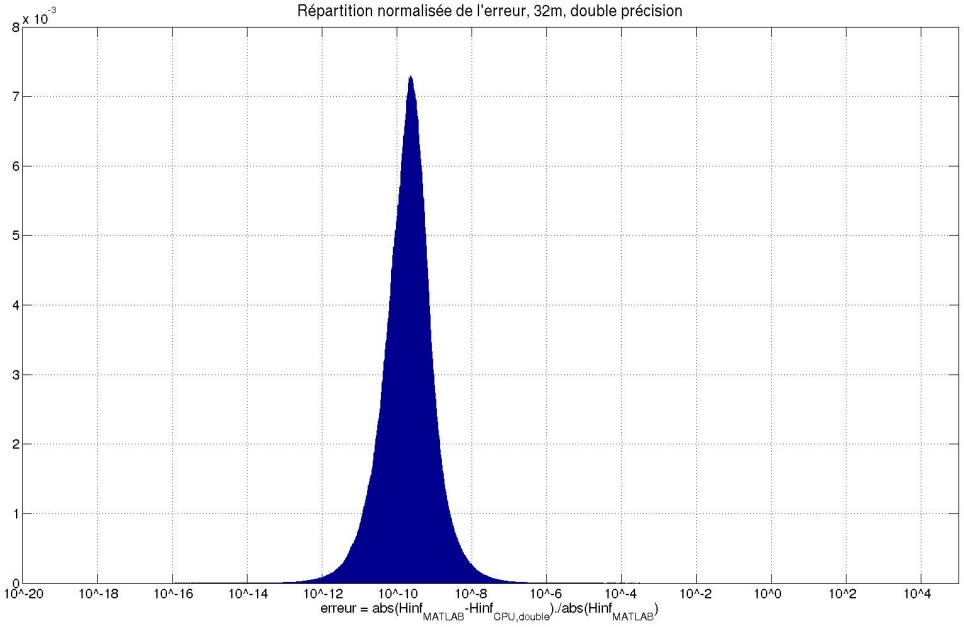


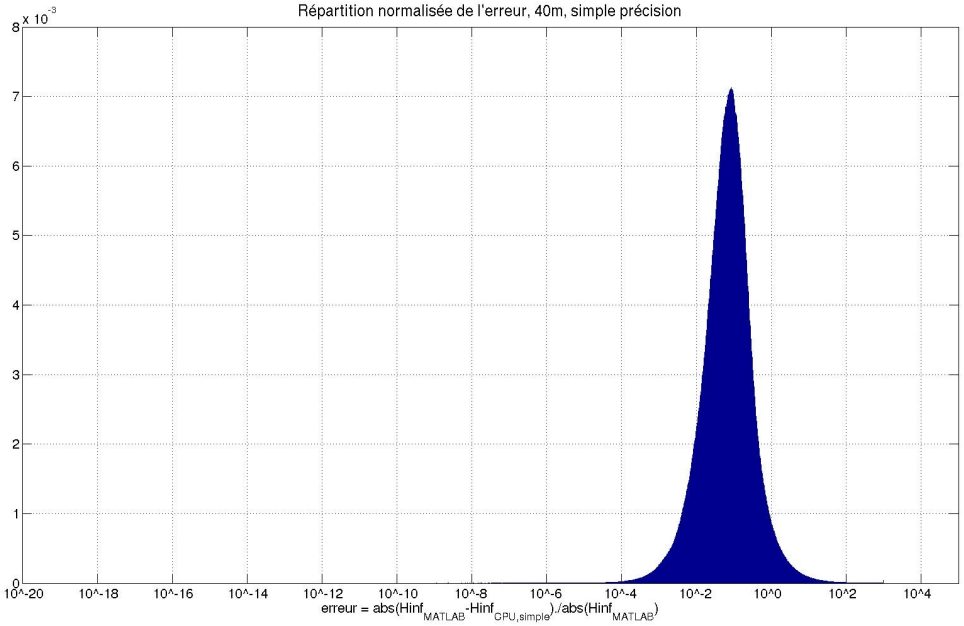


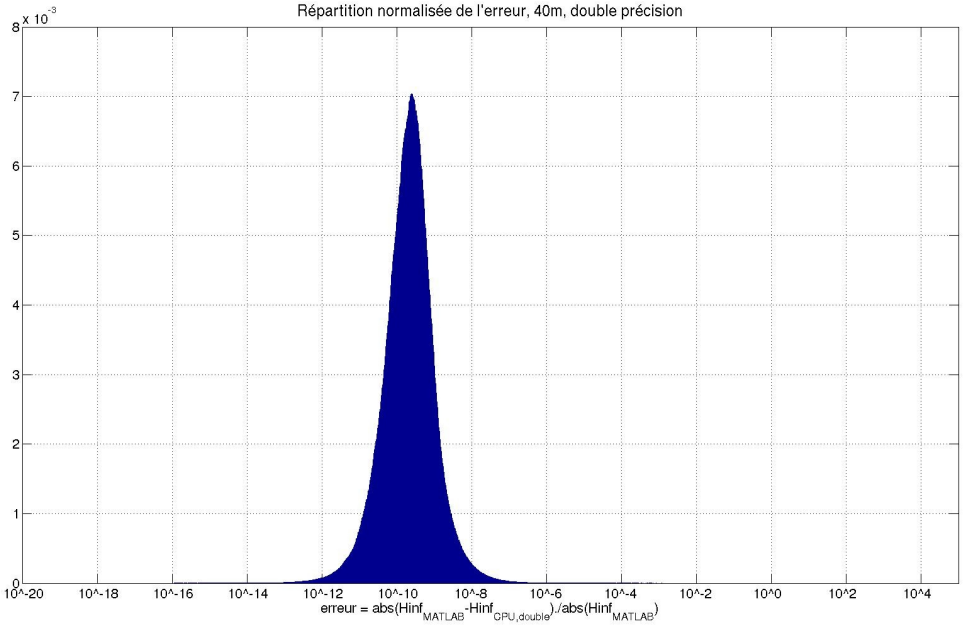

Répartition normalisée de l'erreur, 16m, simple précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Hinf_MATLAB-Hinf_CPU,simple)./abs(Hinf_MATLAB)


Répartition normalisée de l'erreur, 16m, double précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Hinf_MATLAB-Hinf_CPU,double)./abs(Hinf_MATLAB)


Répartition normalisée de l'erreur, 32m, simple précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Hinf_MATLAB-Hinf_CPU,simple)./abs(Hinf_MATLAB)


Répartition normalisée de l'erreur, 32m, double précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Hinf_MATLAB-Hinf_CPU,double)./abs(Hinf_MATLAB)

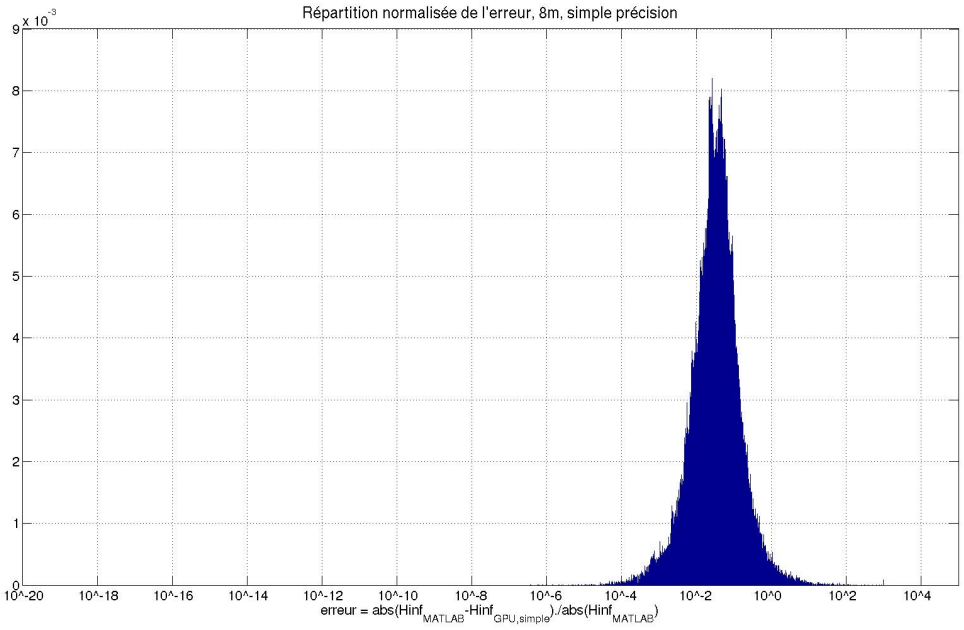

Répartition normalisée de l'erreur, 40m, simple précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Hinf_MATLAB-Hinf_CPU,simple)./abs(Hinf_MATLAB)

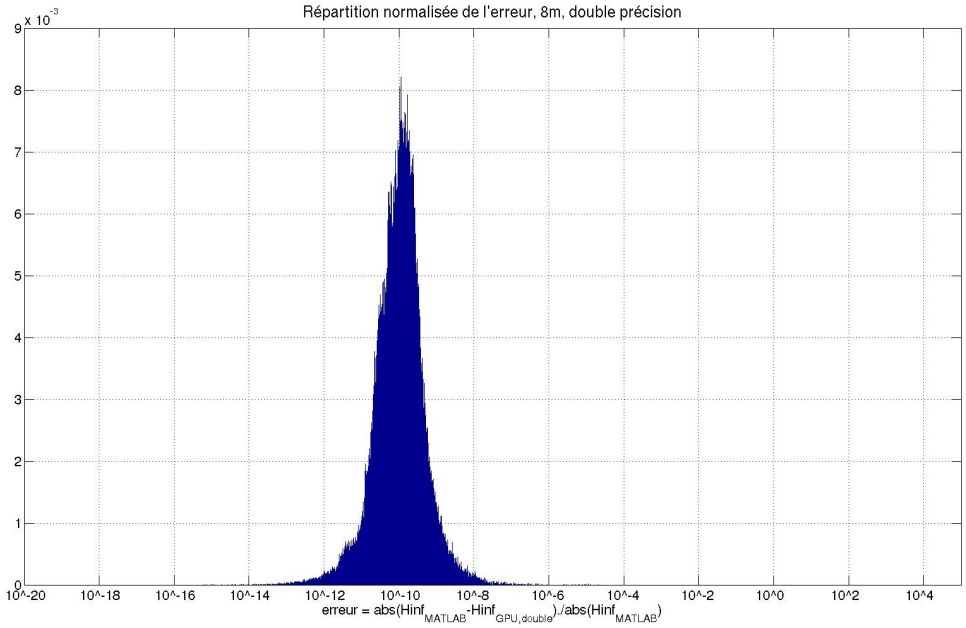

Répartition normalisée de l'erreur, 40m, double précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Hinf_MATLAB-Hinf_CPU,double)./abs(Hinf_MATLAB)

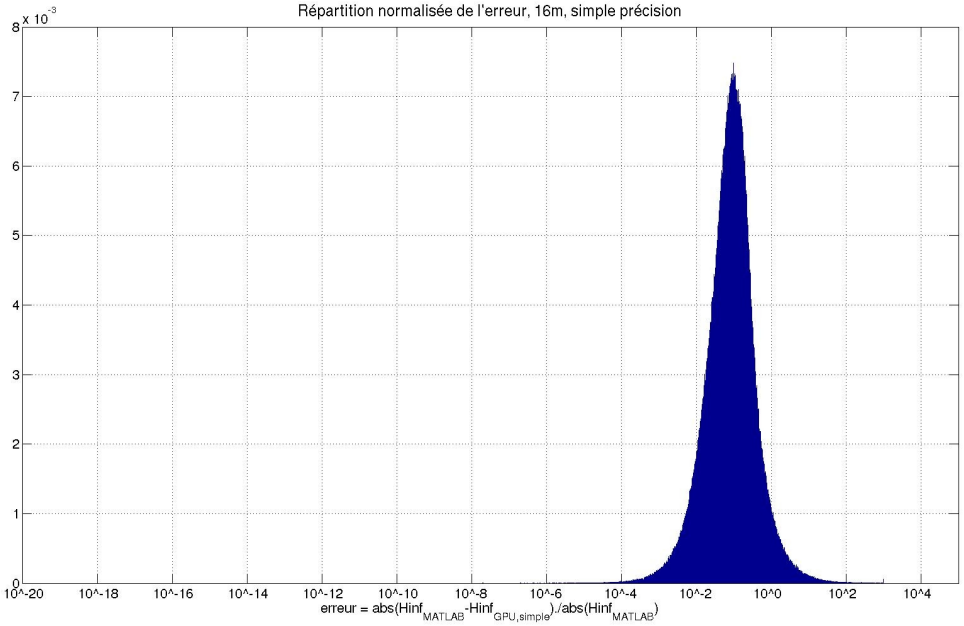
b) Répartition des erreurs relatives sur *H_inf* avec version GPU

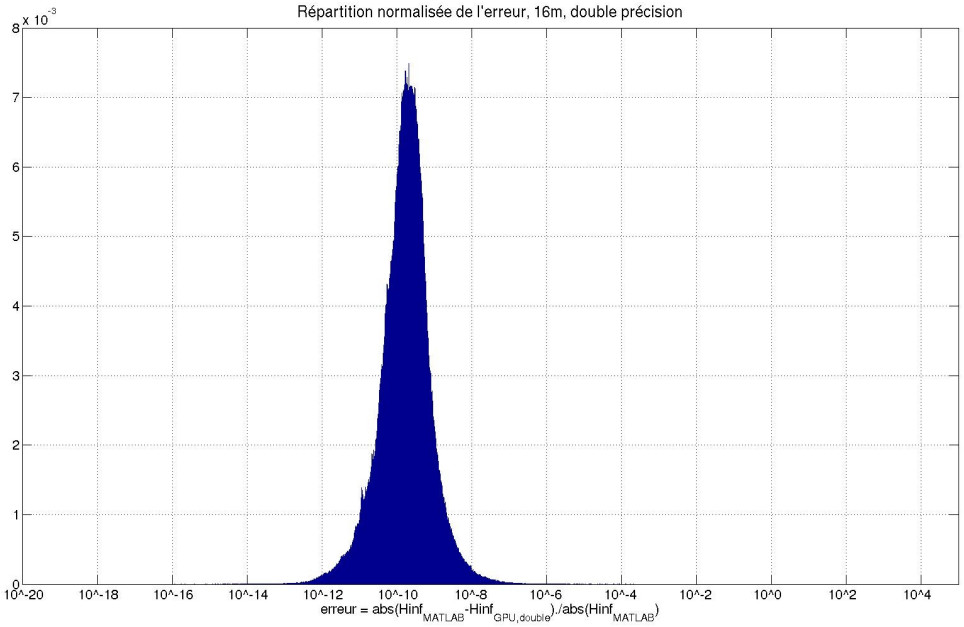
Les figures suivantes montrent de quelle manières sont réparties les erreurs relatives sur les éléments de la matrice *H_inf*, calculée par la version GPU, soit en simple précision, soit en double, pour différents diamètres.

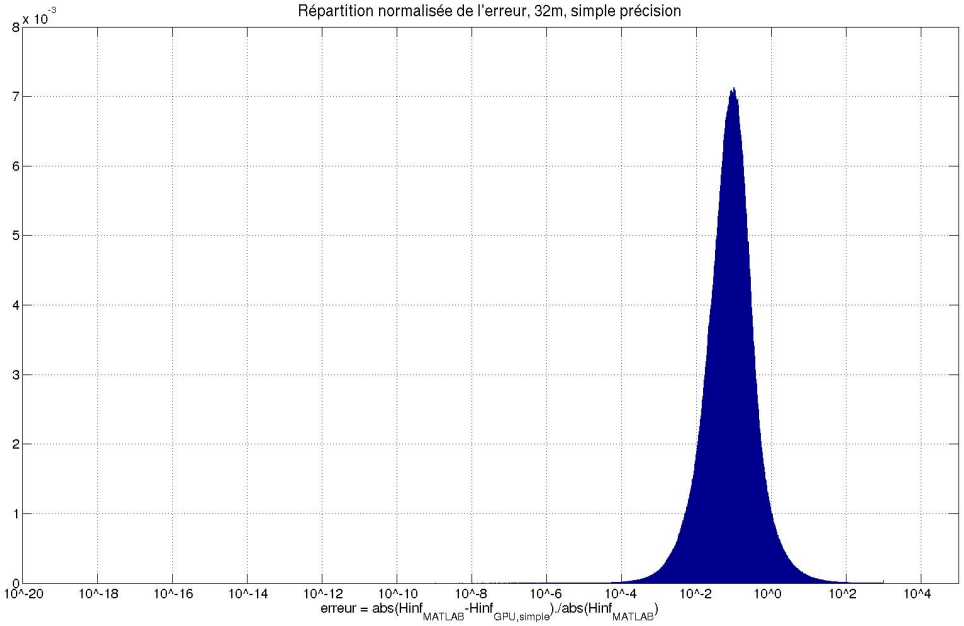
La répartition est normalisée, dans le sens où, pour chaque intervalle d'erreur (en abscisse), le nombre d'occurences correspondantes a été divisé par le nombre total d'éléments. C'est ce rapport *nombre d'occurences / nombre total*, qui est représenté en ordonnée.

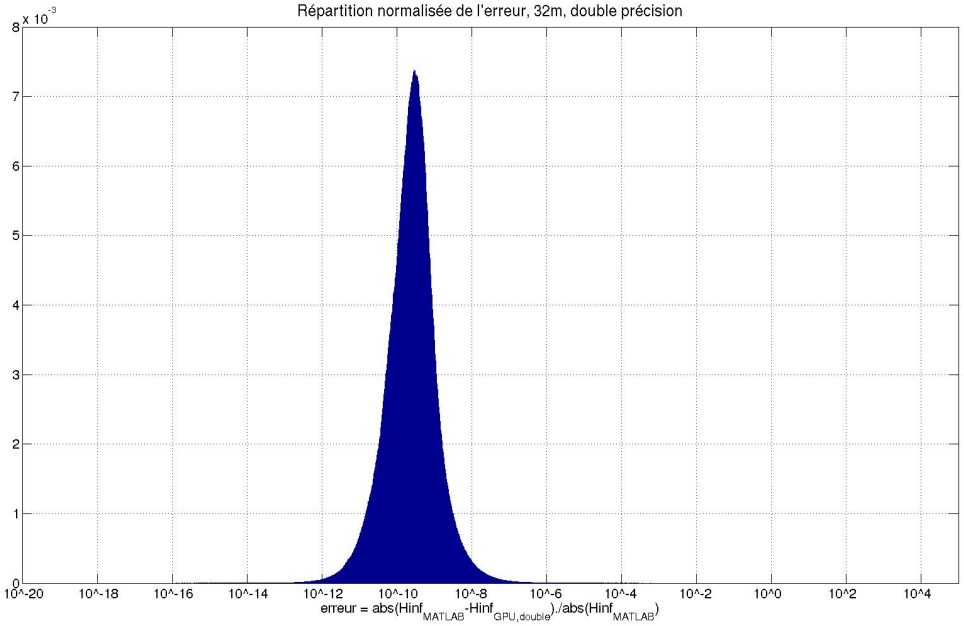


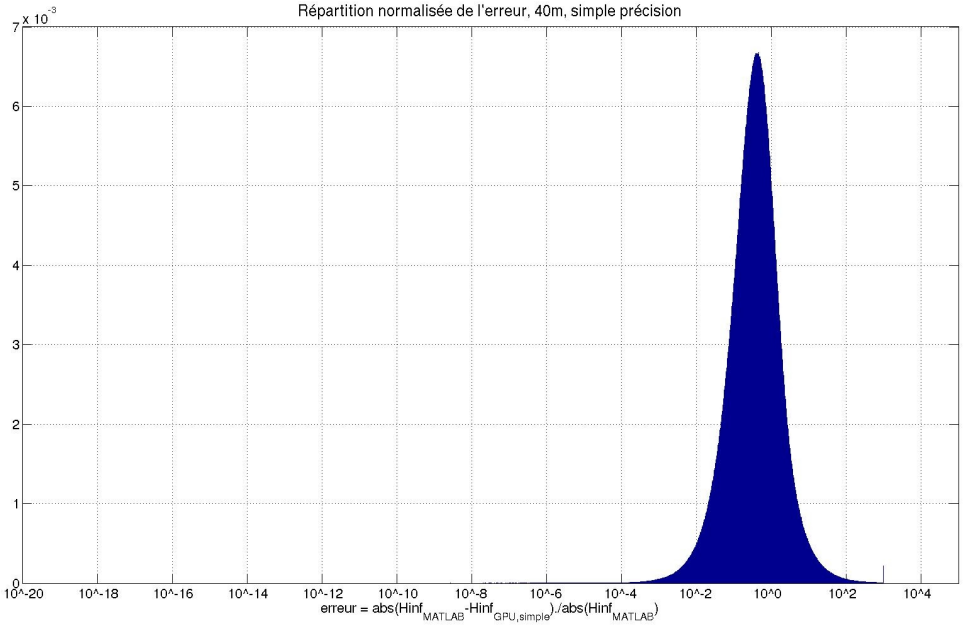


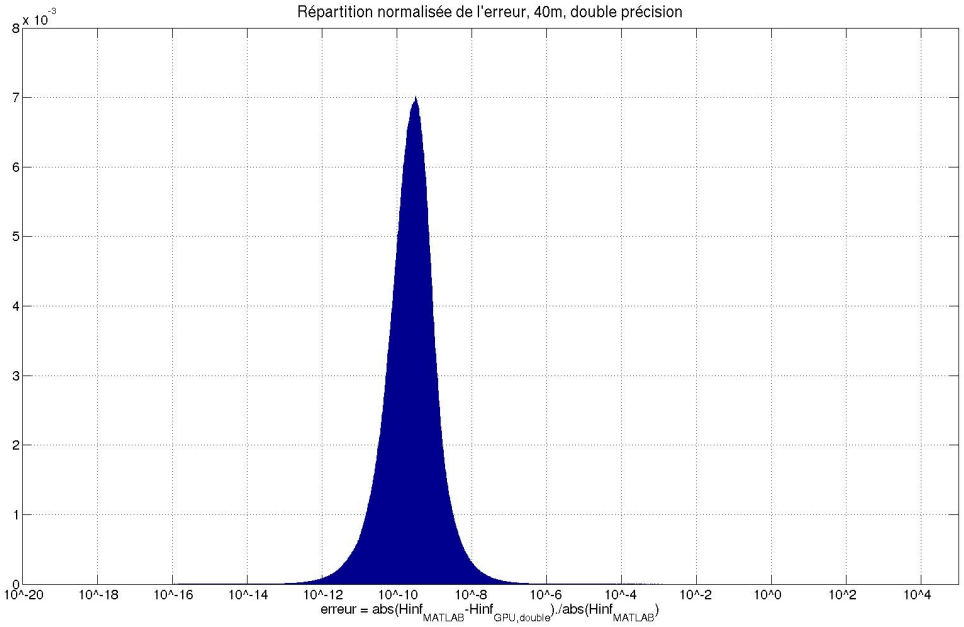

Répartition normalisée de l'erreur, 16m, simple précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Hinf_MATLAB-Hinf_GPU,simple)./abs(Hinf_MATLAB)


Répartition normalisée de l'erreur, 16m, double précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Hinf_MATLAB-Hinf_GPU,double)./abs(Hinf_MATLAB)


Répartition normalisée de l'erreur, 32m, simple précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Hinf_MATLAB-Hinf_GPU,simple)./abs(Hinf_MATLAB)


Répartition normalisée de l'erreur, 32m, double précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Hinf_MATLAB-Hinf_GPU,double)./abs(Hinf_MATLAB)


Répartition normalisée de l'erreur, 40m, simple précision
x 10^-3
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Hinf_MATLAB-Hinf_GPU,simple)./abs(Hinf_MATLAB)


Répartition normalisée de l'erreur, 40m, double précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Hinf_MATLAB-Hinf_GPU,double)./abs(Hinf_MATLAB)

2) Comparaison des résultats entre simple et double précision pour le vecteur *U_k*

Pour le calcul en double précision des vecteurs *U_k*, les résultats obtenus correspondent à ceux obtenus avec MATLAB : une erreur centrée autour 10^{-13}, pour la version CPU comme GPU (voir figures de répartition des erreurs dans la prochaine section).

Etant donné que le calcul de *H_inf* en simple précision donne une matrice complètement différente de celle qu'on est censé obtenir, il n'est pas question de prendre cette matrice pour le calcul des *U_k.*

Cependant lorsqu'on calcule la matrice *H_inf* en double précision et qu'on effectue la suite de l'algorithme (pour calculer *U_k*) en double précision, on s'aperçoit que l'erreur relative obtenue sur les valeurs de Uk est acceptable : autour de 10^{-5} pour la version CPU et autour de 10^{-6} pour la vecrsion GPU(voir figures de répartition des erreurs dans la prochaine section).

Pour la version GPU, environ 99 % des éléments des vecteurs *U_k* (sur 5000 itérations) sont différents de moins de 0,001 % par rapport à ce qu'ils devraient être.

Pour la version CPU, l'erreur relative est un peu plus importante.

		GPU				CPU			
		8m	**16m**	**32m**	**40m**	**8m**	**16m**	**32m**	**40m**
U_k en simple précision à partir de H_inf obtenu en double précision	erreur relative max	38,83%	166,76%	344,86%	400,75%	208,02%	1134,09%	9667,24%	3127,27%
	proportions d'éléments dont erreur < $5x10^{-2}$	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	99,99%	99,99%
	proportions d'éléments dont erreur < 10^{-2}	100,00%	100,00%	100,00%	100,00%	99,98%	99,98%	99,97%	99,96%
	proportions d'éléments dont erreur < 10^{-4}	99,85%	99,88%	99,90%	99,90%	98,48%	97,99%	96,76%	96,07%
	proportions d'éléments dont erreur < 10^{-5}	98,52%	98,84%	98,96%	98,96%	83,76%	72,78%	47,59%	40,17%
	proportions d'éléments dont erreur < 10^{-6}	82,71%	85,30%	85,41%	85,30%	19,51%	11,07%	5,37%	1,36%
	proportions d'éléments dont erreur < 10^{-7}	16,55%	16,52%	15,11%	14,86%	2,00%	1,12%	0,54%	0,44%
	proportions d'éléments dont erreur < 10^{-8}	1,68%	1,67%	1,52%	1,50%	2,00%	0,11%	0,05%	0,04%

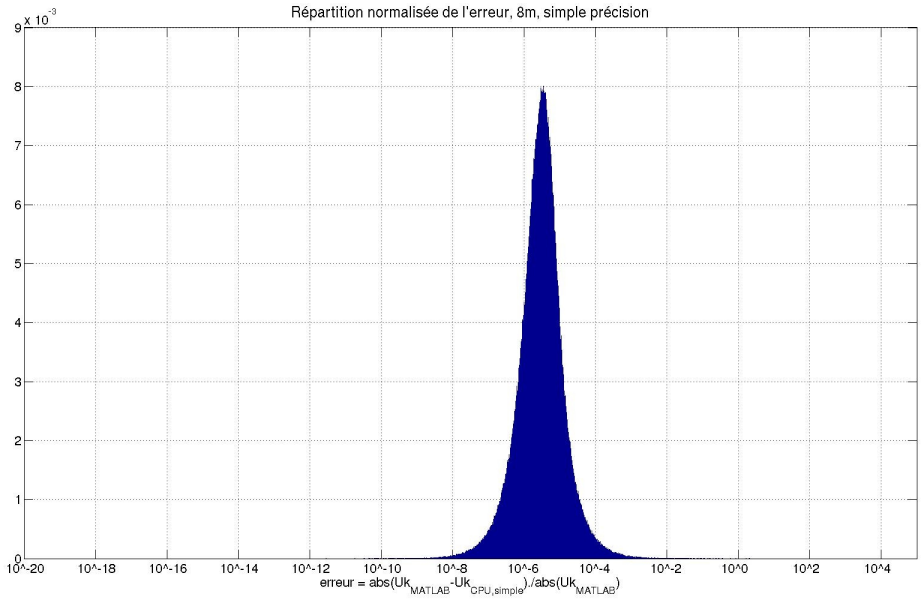
a) Répartition des erreurs relatives sur *U_k* avec version CPU

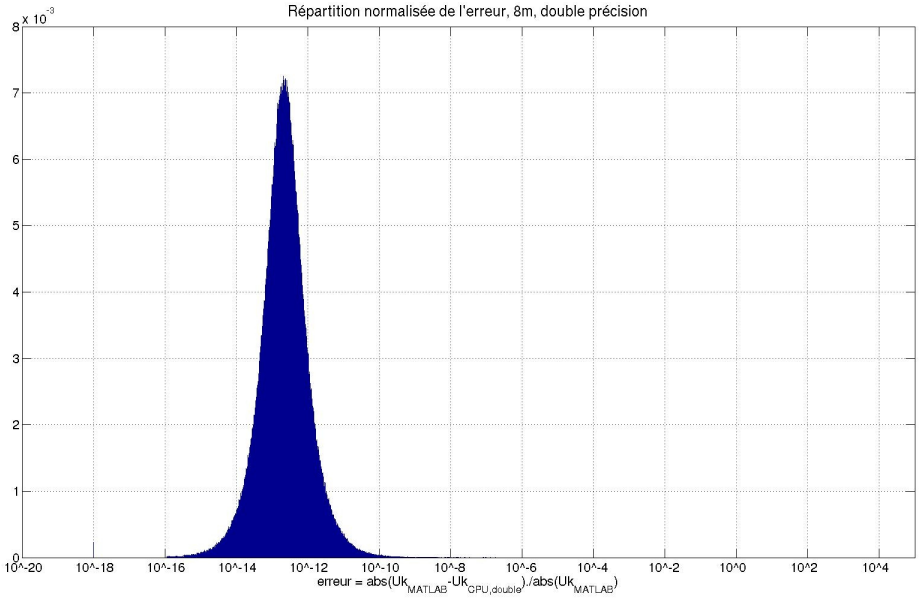
On rappelle que sur les figures de répartitions suivantes, l'erreur relative prise en compte est celle de la matrice composée de 5000 itérations successives de *U_k*.

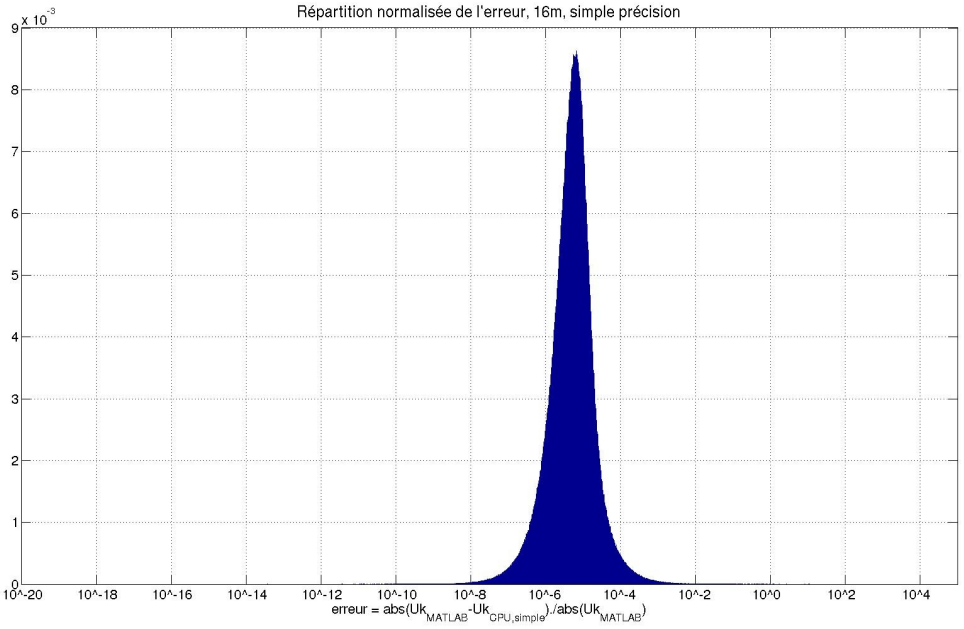
Lorsque la mention « simple précision » apparaît dans le titre de la figure, cela signifie que seul le gain de Kalman est calculé en double précision. Tout le reste de l'algorithme, jusqu'à l'obtention des *U_k* est calculé en simple précision.

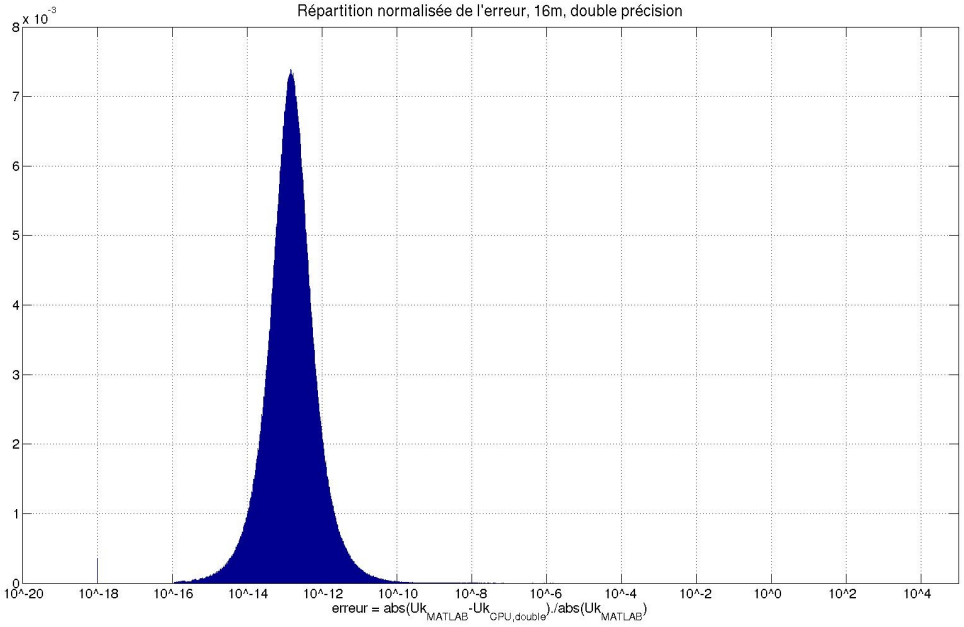
Lorsque la mention «double précision » appaît dans le titre de la figure, cela signifie que tout est calculé en double précision.

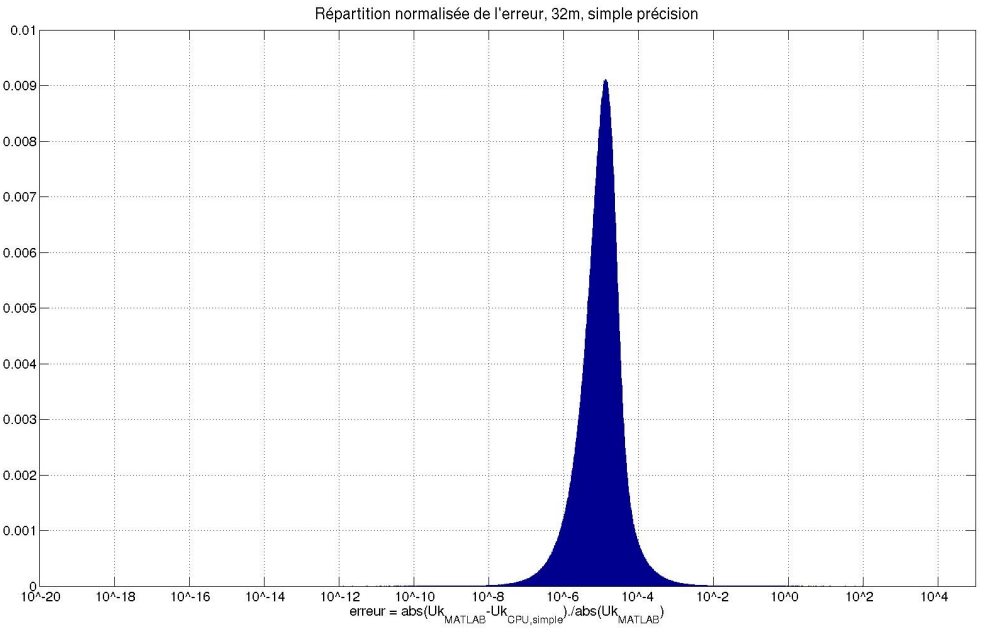
La répartition est normalisée, dans le sens où, pour chaque intervalle d'erreur (en abscisse), le nombre d'occurences correspondantes a été divisé par le nombre total d'éléments. C'est ce rapport *nombre d'occurences / nombre total*, qui est représenté en ordonnée.

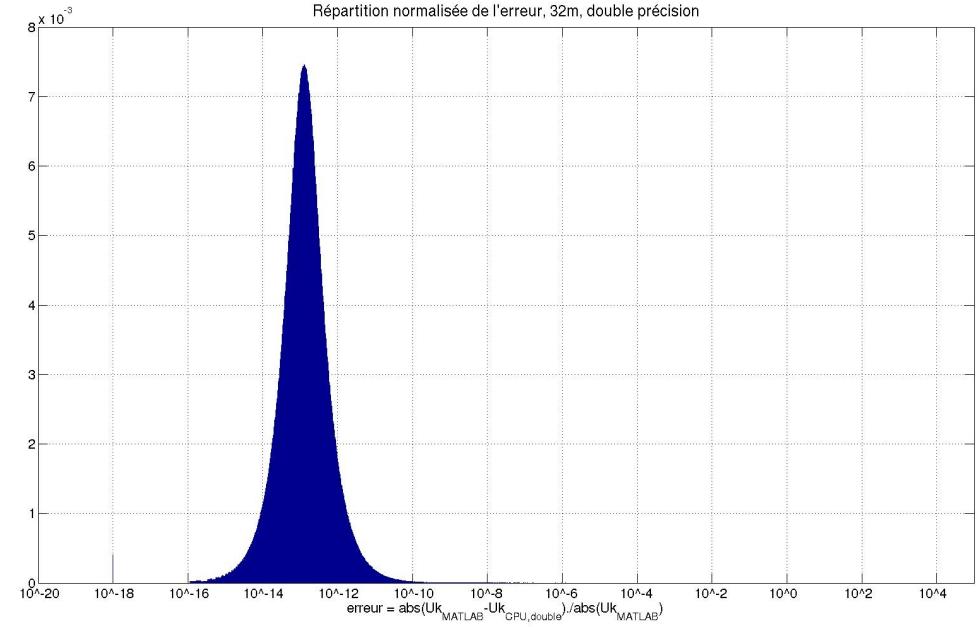


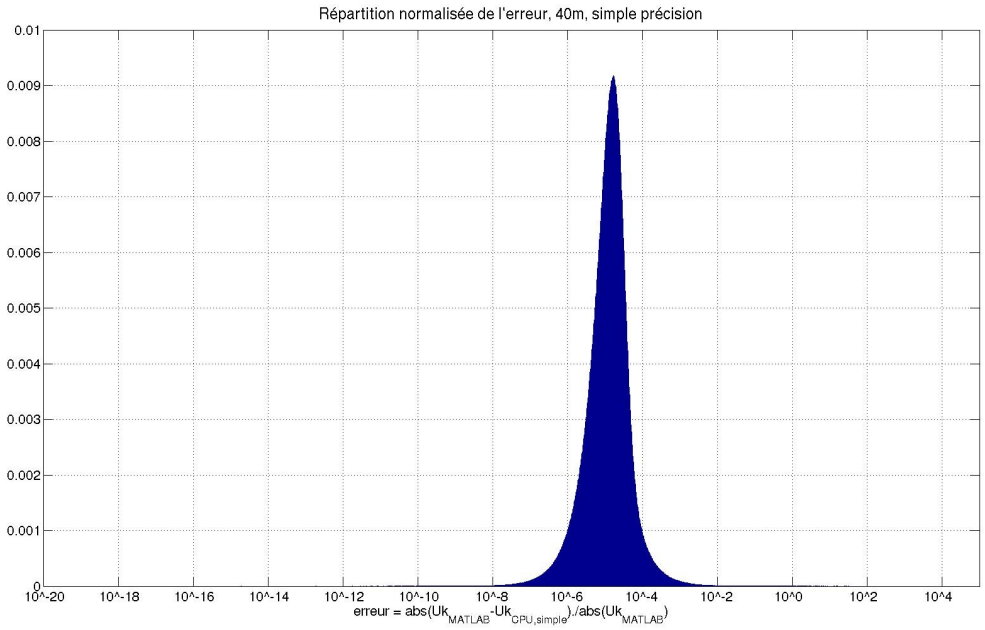


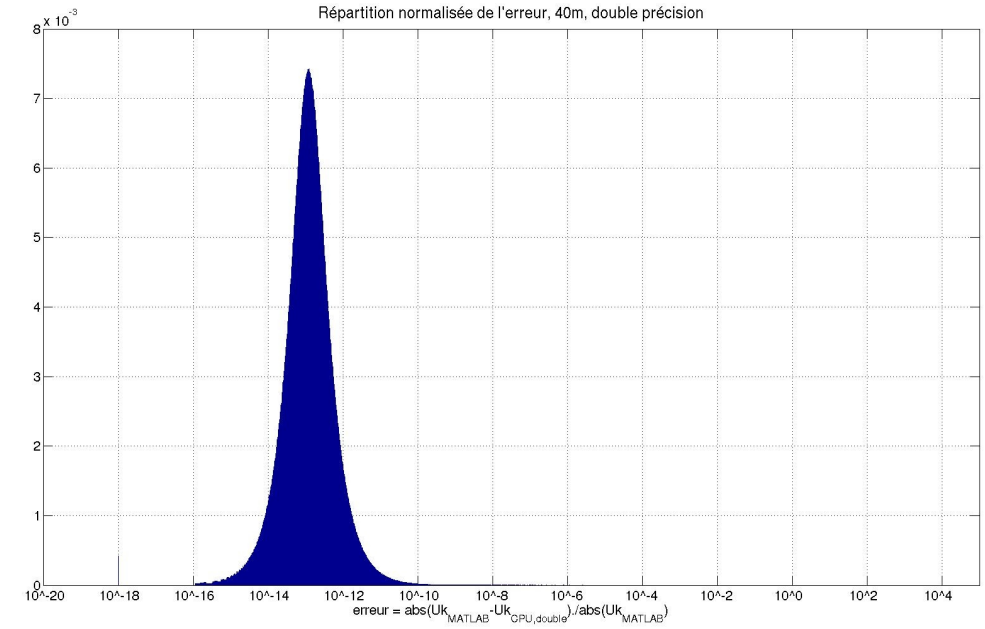

Répartition normalisée de l'erreur, 16m, simple précision
x 10^-3
9
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Uk_MATLAB-Uk_CPU,simple)./abs(Uk_MATLAB)


Répartition normalisée de l'erreur, 16m, double précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Uk_MATLAB-Uk_CPU,double)./abs(Uk_MATLAB)


Répartition normalisée de l'erreur, 32m, simple précision
0.01
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Uk_MATLAB-Uk_CPU,simple)./abs(Uk_MATLAB)


Répartition normalisée de l'erreur, 32m, double précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Uk_MATLAB-Uk_CPU,double)./abs(Uk_MATLAB)


Répartition normalisée de l'erreur, 40m, simple précision
0.01
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Uk_MATLAB-Uk_CPU,simple)./abs(Uk_MATLAB)


Répartition normalisée de l'erreur, 40m, double précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Uk_MATLAB-Uk_CPU,double)./abs(Uk_MATLAB)

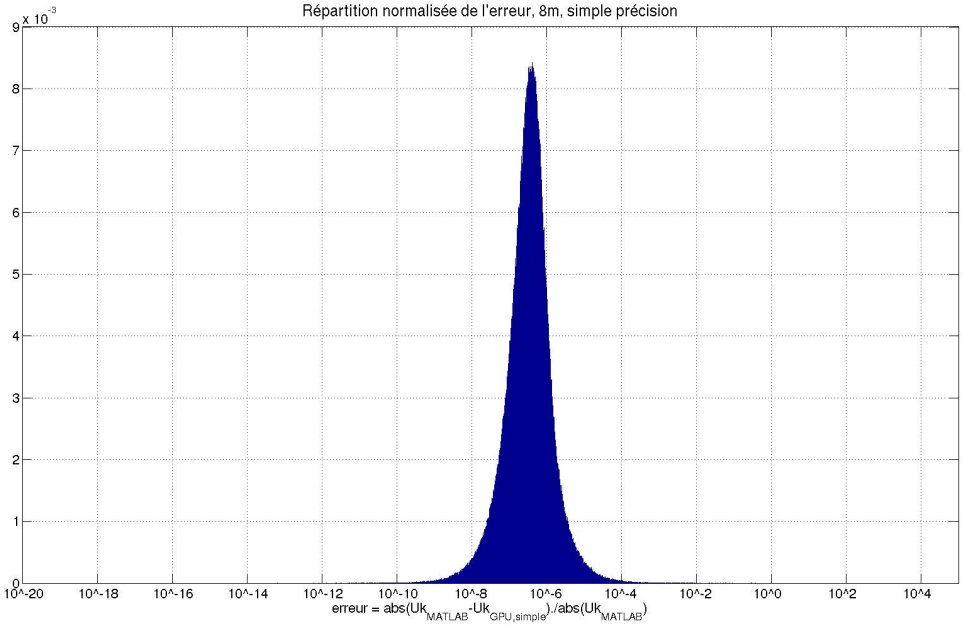
b) Répartition des erreurs relatives sur *U_k* avec version GPU

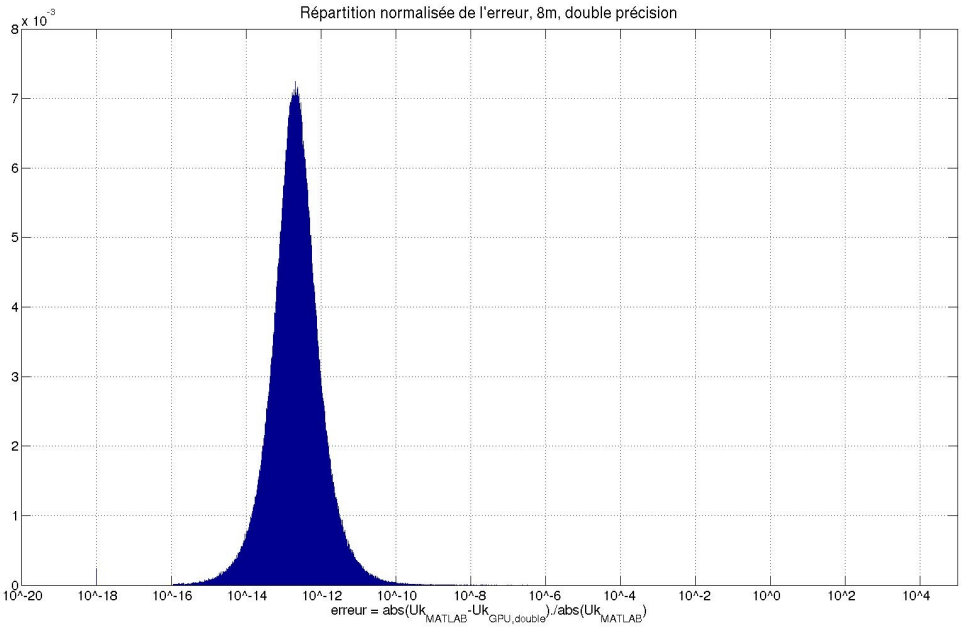
On rappelle que sur les figures de répartitions suivantes, l'erreur relative prise en compte est celle de la matrice composée de 5000 itérations successives de *U_k*.

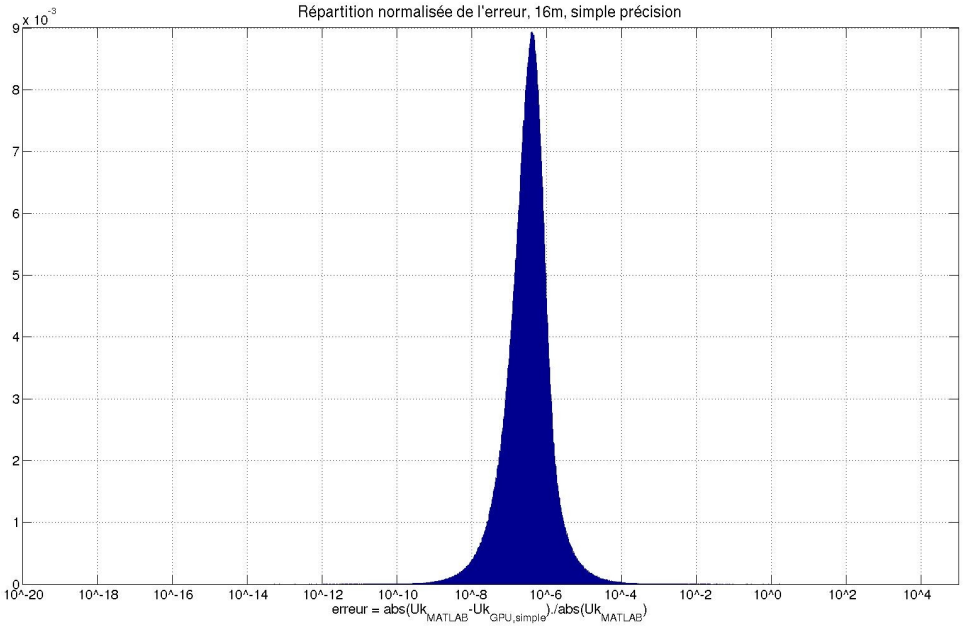
Lorsque la mention « simple précision » apparaît dans le titre de la figure, cela signifie que seul le gain de Kalman est calculé en double précision. Tout le reste de l'algorithme, jusqu'à l'obtention des *U_k* est calculé en simple précision.

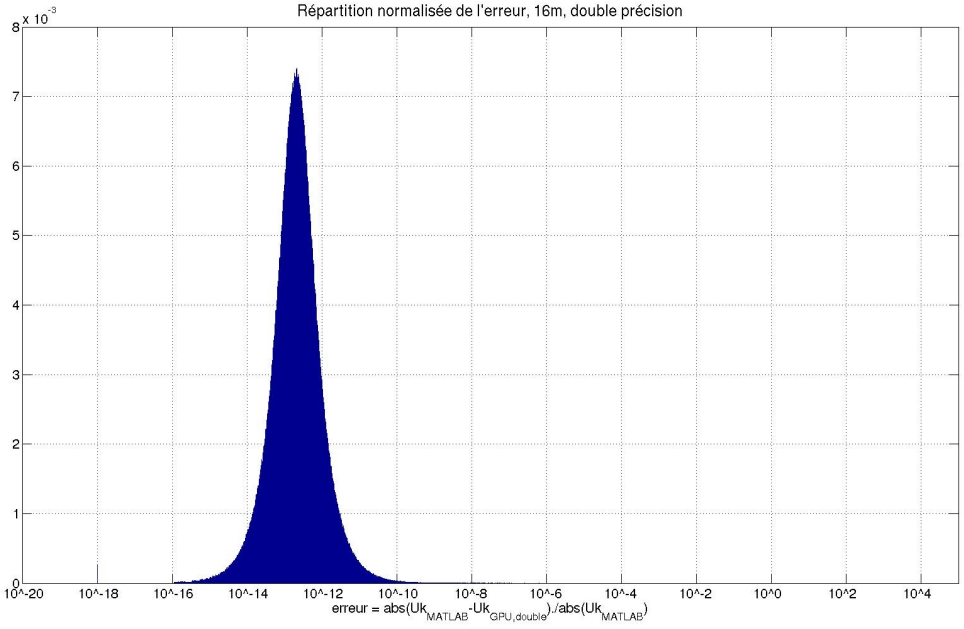
Lorsque la mention «double précision » appaît dans le titre de la figure, cela signifie que tout est calculé en double précision.

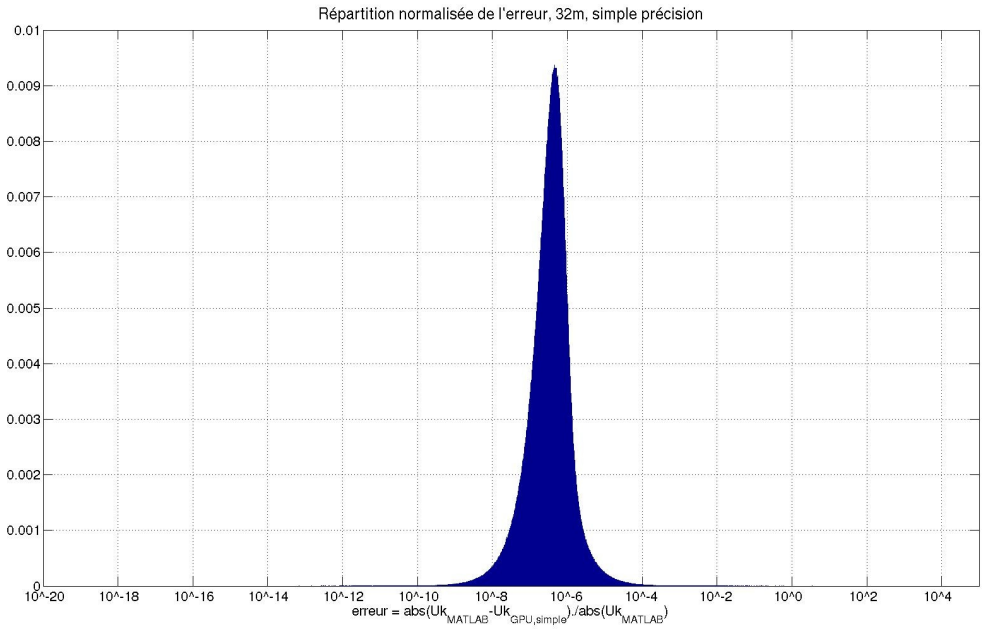
La répartition est normalisée, dans le sens où, pour chaque intervalle d'erreur (en abscisse), le nombre d'occurences correspondantes a été divisé par le nombre total d'éléments. C'est ce rapport *nombre d'occurences / nombre total*, qui est représenté en ordonnée.

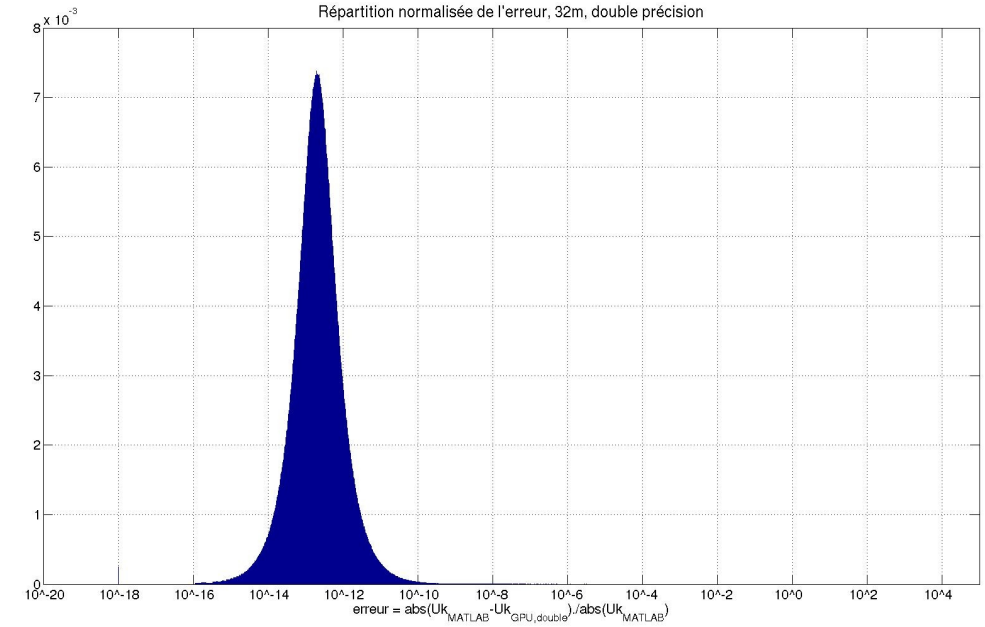


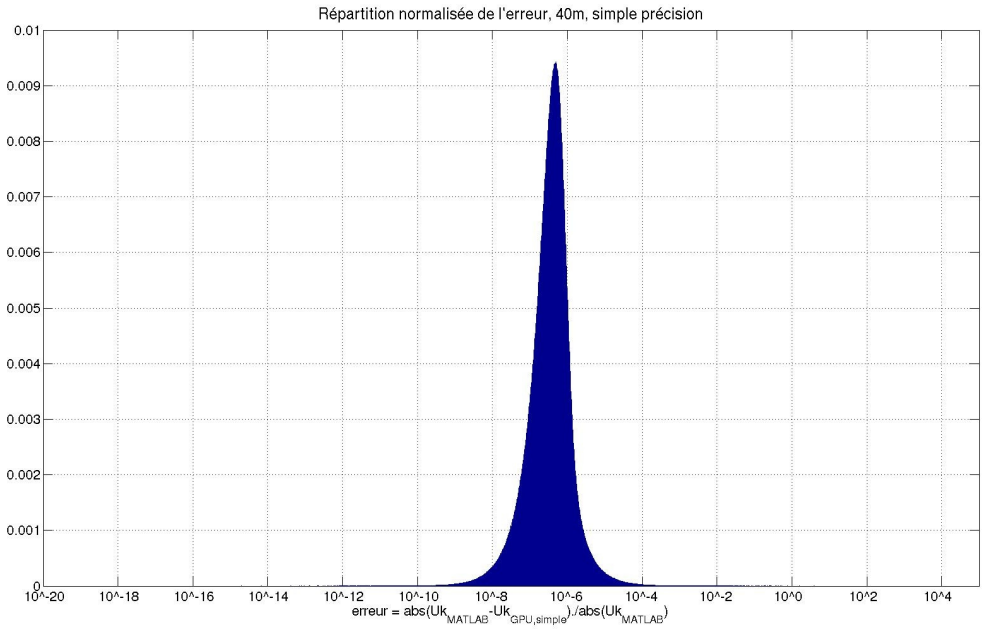


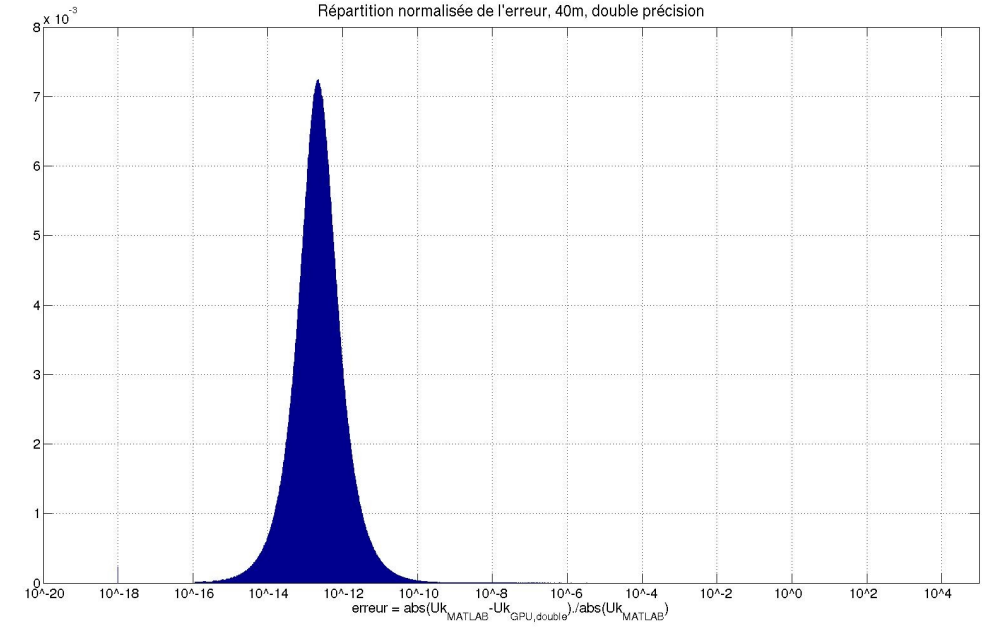

Répartition normalisée de l'erreur, 16m, simple précision
x 10^-3
9
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Uk_MATLAB-Uk_GPU,simple)./abs(Uk_MATLAB)


Répartition normalisée de l'erreur, 16m, double précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Uk_MATLAB-Uk_GPU,double)./abs(Uk_MATLAB)


Répartition normalisée de l'erreur, 32m, simple précision
0.01
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Uk_MATLAB-Uk_GPU,simple)./abs(Uk_MATLAB)


Répartition normalisée de l'erreur, 32m, double précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Uk_MATLAB-Uk_GPU,double)./abs(Uk_MATLAB)


Répartition normalisée de l'erreur, 40m, simple précision
0.01
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Uk_MATLAB-Uk_GPU,simple)./abs(Uk_MATLAB)


Répartition normalisée de l'erreur, 40m, double précision
x 10^-3
8
7
6
5
4
3
2
1
0
10^-20
10^-18
10^-16
10^-14
10^-12
10^-10
10^-8
10^-6
10^-4
10^-2
10^0
10^2
10^4
erreur = abs(Uk_MATLAB-Uk_GPU,double)./abs(Uk_MATLAB)

3) Comparaison des temps d'exécution entre simple et double précision pour le calcul des vecteurs *U_k*

Dans le tableau suivant, on compare l'influence du calcul en simple précision sur le temps de calcul, pour la meilleure version CPU (matrices creuses column-major) et GPU (matrices creuses row-major).

Dans le cas de la version GPU, plus le diamètre est grand, plus la précision utilisée à une influence sur le temps de calcul. On observe ainsi un rapport de temps de 1,50 entre la version simple et double, pour de grands diamètres.

Pour la version CPU, le rapport des temps est légèrement supérieur pour de grands diamètres.

Le gain entre les versions GPU et CPU en simple précision est de 3,43 pour les grands diamètres, ce qui est légèrement inférieur à celui en double précision (rapport de 4,30 pour 32m et 3,95 pour 40m)

Temps de calcul des tensions U_k à partir des pentes Y_k

zonal, AR1, Matrices Creuses		GPU row-major			CPU col-major			
		temps double precision (s) (5000 itérations)	temps simple precision (s) (5000 itérations)	temps double / temps simple	temps double precision(s) (5000 itérations)	temps simple precision (s) (5000 itérations)	temps double / temps simple	temps simple CPU / temps simple GPU
8m	op1	0,306	0,308	0,99	0,0504	0,0465	1,08	0,15
	op2	0,769	0,679	1,13	0,759	0,108	7,03	0,16
	op3	0,365	0,354	1,03	0,302	0,3	1,01	0,85
	Total 2	**1,75**	**1,7**	**1,03**	**1,13**	**0,472**	**2,39**	**0,28**
16m	op1	0,486	0,374	1,30	0,235	0,223	1,05	0,60
	op2	1,65	1,27	1,30	3,15	1,42	2,22	1,12
	op3	0,516	0,44	1,17	2,57	2,25	1,14	5,11
	Total 2	**3,2**	**2,51**	**1,27**	**6**	**3,92**	**1,53**	**1,56**
32m	op1	0,578	0,538	1,07	0,85	0,814	1,04	1,51
	op2	13,8	8,65	1,60	61,1	28,8	2,12	3,33
	op3	1,25	0,991	1,26	13,1	11,4	1,15	11,50
	Total 2	**17,5**	**12**	**1,46**	**75,2**	**41,1**	**1,83**	**3,43**
40m	op1	0,626	0,531	1,18	1,42	1,09	1,30	2,05
	op2	32,3	20,7	1,56	119	64,1	1,86	3,10
	op3	1,85	1,4	1,32	23,9	18,6	1,28	13,29
	Total 2	**36,7**	**24,4**	**1,50**	**145**	**83,8**	**1,73**	**3,43**

op1 : Y_kskm1 = D_Mo*(X_kskm1(nb_az+1:end) – N_Act*U_km2);
op2 : A1*(X_kskm1 + H_inf*(Y_k – Y_kskm1));
op3 : U_k = PROJ*(X_kp1sk(1:nb_az)-mean(X_kp1sk(1:nb_az))*ones(nb_az,1));
Total2 correspond au temps total pour calculer U_k à partir de Y_k

VIII) Bilan de la version standalone

Il semble nécessaire d'effectuer le calcul de la matrice *H_inf* en double précision.
Une fois que la matrice *H_inf* est calculée, il est possible d'effectuer le reste de l'algorithme en simple précision.

La version CPU est plus rapide si les matrices creuses sont stockées en column-major.
La version GPU est plus rapide si les matrices creuses sont stockées en row-major.

Le rapport entre le temps d'exécution de la meilleure version CPU et la meilleure version GPU est de :

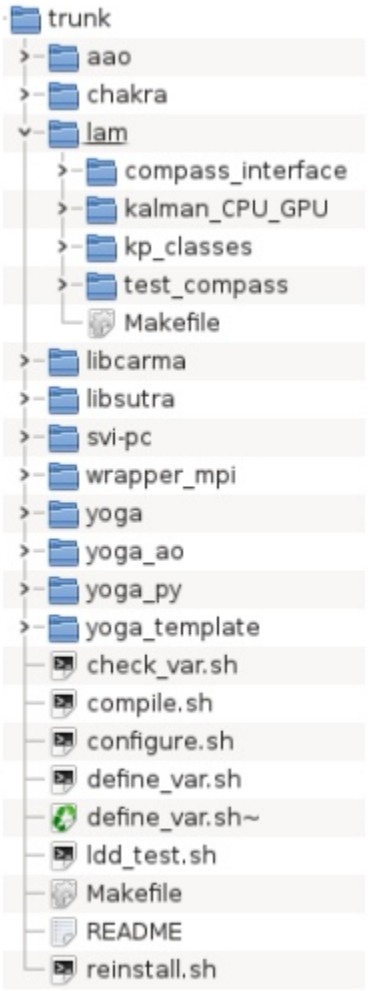
- 4,30 pour 32m en double précision
- 3,95 pour 40m en double précision
- 3,43 pour 32m et 40m en simple précision

Partie B : Intégration à la plateforme COMPASS

I) Arborescence

Le répertoire *$COMPASS_ROOT_DIR/trunk/lam* contient tous les fichiers nécessaires pour pouvoir utiliser le filtre de Kalman dans COMPASS :

1) Le répertoire lam/

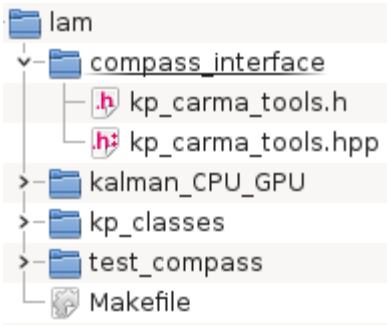


Ce répertoire contient quatre sous-répertoires :

- kp_classes : contient les classes représentant des matrices (pleines ou creuses), des vecteurs et d'autres fichiers définissant des opérations basiques sur CPU ou GPU.
- kalman_CPU_GPU : contient tous les fichiers sources du filtre de Kalman proprement dit. Ces fichiers dont le nom commence par kp_kalman_core, utilise les classes contenues dans les fichiers du répertoire kp_classes.
- compass_interface : contient les fichiers kp_carma_tools.h et kp_carma_tools.hpp, permettant la conversion entre les objets de libcarma (carma_obj, carma_host_obj) et ceux de kp_classes (kp_cu_vector, kp_cu_matrix, kp_vector, kp_matrix).
- test_compass : répertoire pour effectuer des tests en batch avec COMPASS pour Kalman ou LS.

Ce répertoire contient également un Makefile, permettant la compilation des fichiers sources des 3 sous-répertoires, nécessaires pour l'utilisation dans COMPASS.

2) Le répertoire compass_interface



Ce répertoire contient deux fichiers : kp_carma_tools.h et kp_carma_tools.hpp. Ces fichiers définissent les fonctions suivantes :

- *template<typename T, typename U>*

void kp_carma_host_obj_to_kp_matrix(carma_host_obj<T>& ch, kp_matrix<U>& k)
Copie CPU vers CPU, d'un objet carma_host_obj vers un objet kp_matrix.

- *template<typename T, typename U>*

void kp_carma_obj_to_kp_matrix(carma_obj<T>& c, kp_matrix<U>& k)
Copie GPU vers CPU, d'un objet carma_obj vers un objet kp_matrix.

- *template<typename T, typename U>*

void kp_carma_host_obj_to_kp_vector(carma_host_obj<T>& ch, kp_vector<U>& k)
Copie CPU vers CPU, d'un objet carma_host_obj vers un objet kp_vector.

- *template<typename T, typename U>*

void kp_carma_obj_to_kp_vector(carma_obj<T>& c, kp_vector<U>& k)
Copie GPU vers CPU, d'un objet carma_obj vers un objet kp_vector.

- *template<typename T, typename U>*

void kp_kp_vector_to_carma_obj(const kp_vector<T>& k, carma_obj<U>& c)
Copie CPU vers GPU d'un objet kp_vector vers un objet carma_obj.

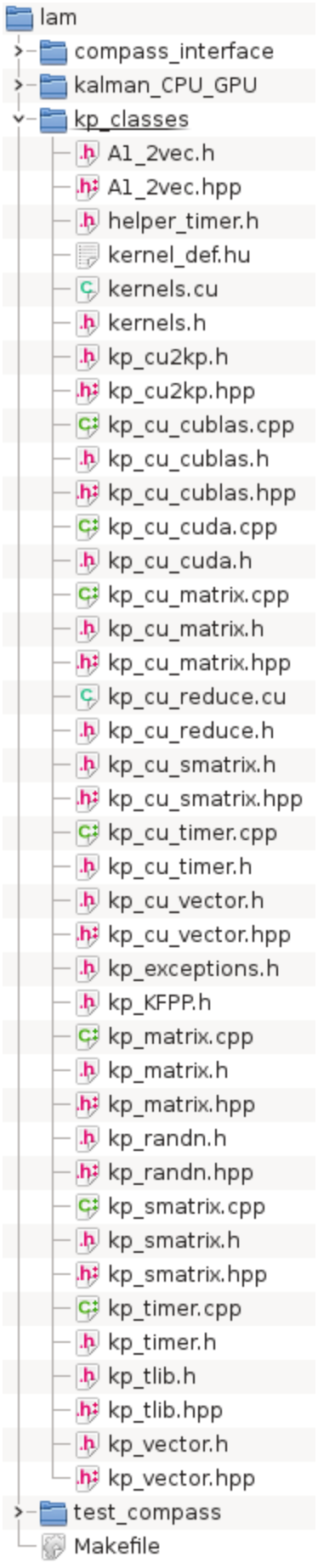
- *template<typename T, typename U>*

void kp_kp_cu_vector_to_carma_obj(const kp_cu_vector<T>& cu_k, carma_obj<U>& c)
Copie GPU vers GPU d'un objet kp_cu_vector vers un objet carma_obj.

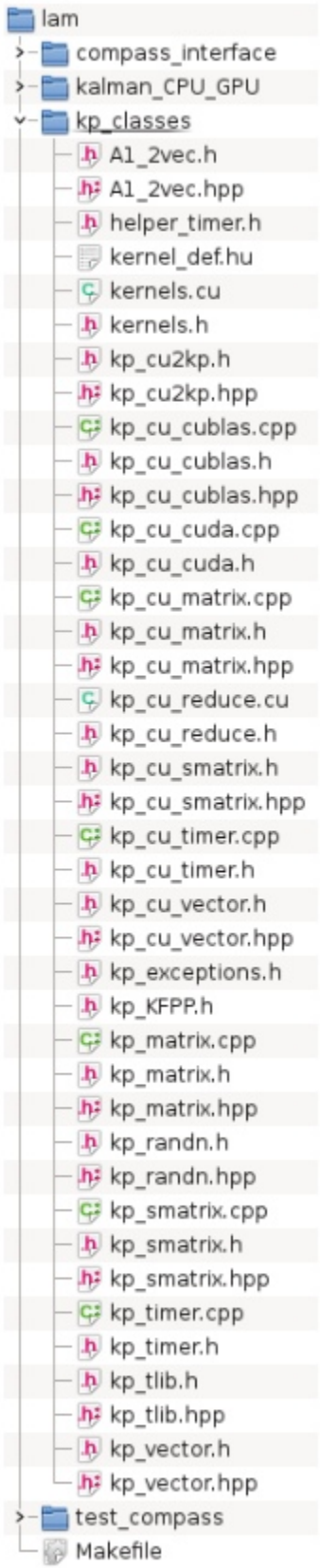
- *template<typename T, typename U>*

void kp_carma_obj_to_kp_cu_vector(const carma_obj<T>& c, kp_cu_vector<U>& cu_k)
Copie GPU vers GPU d'un objet carma_obj vers un objet kp_cu_vector.

Ces conversions permettent la conversion entre les instances de carma_obj, carma_host_obj de libcarma et celles de kp_vector, kp_matrix, kp_cu_vector, kp_cu_matrix. Elles sont utilisées dans libsutra/src.cpp/sutra_controller_kalman.cpp, dans les méthodes de la classe sutra_controller_kalman qui font appel auxméthodes de la classe kp_kalman_core, utilisant les classes kp_vector, kp_matrix, kp_cu_vector et kp_cu_matrix.
Ces classes étant des classes templates, il est possible d'utiliser ces fonctions aussi bien avec deux objets en double précision, avec deux objets en simple précision ou avec un objet en simple et l'autre en double précision.


lam
compass_interface
kalman_CPU_GPU
kp_classes
A1_2vec.h
A1_2vec.hpp
helper_timer.h
kernel_def.hu
kernels.cu
kernels.h
kp_cu2kp.h
kp_cu2kp.hpp
kp_cu_cublas.cpp
kp_cu_cublas.h
kp_cu_cublas.hpp
kp_cu_cuda.cpp
kp_cu_cuda.h
kp_cu_matrix.cpp
kp_cu_matrix.h
kp_cu_matrix.hpp
kp_cu_reduce.cu
kp_cu_reduce.h
kp_cu_smatrix.h
kp_cu_smatrix.hpp
kp_cu_timer.cpp
kp_cu_timer.h
kp_cu_vector.h
kp_cu_vector.hpp
kp_exceptions.h
kp_KFPP.h
kp_matrix.cpp
kp_matrix.h
kp_matrix.hpp
kp_randn.h
kp_randn.hpp
kp_smatrix.cpp
kp_smatrix.h
kp_smatrix.hpp
kp_timer.cpp
kp_timer.h
kp_tlib.h
kp_tlib.hpp
kp_vector.h
kp_vector.hpp
test_compass
Makefile

3) Le répertoire kp_classes



Ce répertoire contient tous les fichiers définissant les fonctions et classes utilisées par celles du filtre de Kalman (kp_kalman_core*). Voici les différents fichiers de ce répertoire et leurs fonctionnalités :

- A1_2vec.h/A1_2vec.hpp : permet d'obtenir les 2 vecteurs atur et btur, à partir de la matrice creuse A1.

- kernels.h/kernels.cu/kernel_def.hu

Cette classe définit tous les kernels personnalisés qui sont exécutés sur GPU : copie de données, opérations de base (additions, soustractions, multiplications, divisions), opérations spécialisées sur des matrices (obtention d'une matrice pleine à partir d'une matrice creuse, addition élément par élément de la diagonale d'une matrice avec un vecteur, copie d'une sous-matrice dans une matrice plus grande, etc.). Le fichier kernels.cu est compilé avec nvcc afin d'obtenir un fichier kernels.o qui est ensuite inclus lors de la compilation de kalman.

Le fichier kernels.h contient les prototypes de toutes les fonctions de mise en forme pour l'exécution des kernels. Toutes ces fonctions peuvent être directement appelées dans un programme en C++. Ce sont des fonctions plutôt bas niveau, puisqu'elles prennent comme arguments des adresses de tableaux (représentant des matrices ou vecteurs), des entiers et des réels (float ou double selon le type choisit lors de la compilation).

Le fichier kernels.cu est composé de trois grandes parties :

- les prototypes des kernels,
- les définitions des fonctions de mise en forme pour l'exécution des kernels, définissant notamment le nombre de threads par bloc et le nombre de blocs par grille,
- le définition des kernels.

Le fichier kernel_def.cu, inclus la fin du fichier kernels.cu contient la liste des prototypes spécialisés, nécessaires pour pouvoir utiliser les classes templates avec CUDA.

- kp_cu_cuda.h/kp_cu_cuda.cpp

Ces fichiers permettent d'allouer, de copier ou de libérer de la mémoire sur GPU, en utilisant les fonctions *cudaMalloc*, *cudaMemcpy*, *cudaMemset* et *cudaFree*, en vérifiant que l'action demandée s'est correctement effectuée.

- kp_timer.h/kp_timer.cpp

Classe permettant de mesurer la durée d'exécution d'une portion de code en C++, à l'aide de ses méthodes *reset*, *start*, *pause* et *rez*.

- kp_cu_timer.h/kp_cu_timer.cpp

Classe permettant de mesurer la durée d'exécution d'une portion de code en C++ contenant du code CUDA, à l'aide de ses méthodes *reset*, *start*, *pause* et *rez*.

- kp_KFPP.h

Défini le type *KFPP* (pour Kalman Floating Point Precision) en tant que double ou float selon la définition ou non de la variable de compilation *KP_DOUBLE*.

- kp_cu_reduce.h/kp_cu_reduce.cu

Prend comme argument un objet *kp_cu_vector*, passé par référence, et retourne la somme de tous les éléments de ce vecteur, en utilisant l'algorithme de réduction pour GPU de la bibliothèque Thrust.

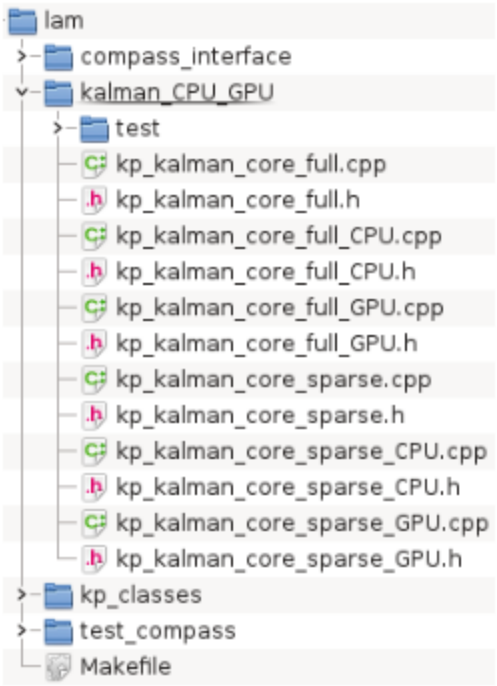
- kp_cu2kp.h/kp_cu2kp.hpp

Ces fichiers définissent des fonctions (*kp_cu2kp_vector, kp_cu2kp_matrix, kp_cu2kp_smatrix*), qui permettent de convertir un objet sur GPU (de classe *kp_cu_vector, kp_cu_matrix* ou *kp_cu_smatrix*) en un objet sur CPU (de classe *kp_vector, kp_matrix* ou *kp_smatrix*) .

- kp_tlib.h/kp_tlib.hpp

Comprend la fonction *init_smatrix_from_larger_smatrix*, permettant d'initialiser une sous-matrice creuse, contenue dans une matrice creuse plus grande.

4) Le répertoire kalman_CPU_GPU

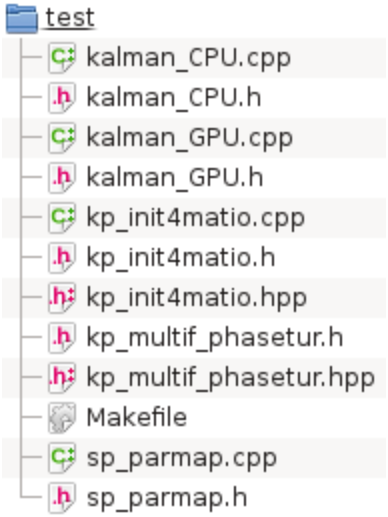


Ce répertoire contient les fichiers commençant par kp_kalman_core, qui définissent les classes représentant le filtre de Kalman.

Ces classes font intervenir des objets comme des matrices creuses (kp(_cu)_smatrix), pleines (kp(_cu)_matrix) et vecteurs (kp(_cu)_vector), définis dans le répertoire kp_classes. Elles font également intervenir des fonctions définies dans des fichiers du répertoire kp_classes, utilisant aussi ces matrices et vecteurs.

5) Le répertoire test

Le répertoire *test*, situé dans *kalman_CPU_GPU* correspond à la version standalone du filtre de Kalman.

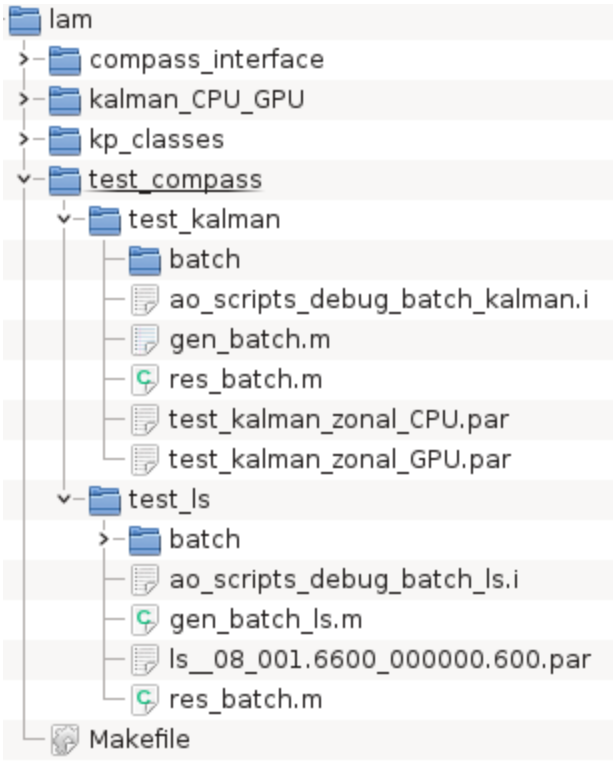


Les classes kalman_CPU et kalman_GPU gèrent toute la boucle d'OA et appellent à chaque itération une des classes kp_kalman_core*

La classe kp_init4matio permet d'obtenir un vecteur, une matrice creuse ou pleine à partir d'un fichier au format MATLAB (.mat).

La kp_multif_phasetur permet d'obtenir le vecteur de phase turbulente de l'itération en cours à partir d'une matrice composée de l'ensemble des vecteurs de phase turbulente sur un nombre d'itération donnée (5000 généralement).

6) Le répertoire test_compass



Le répertoire test_compass est utilisé pour effectuer des tests en batch sur COMPASS avec le controlleur Kalman (sous-répertoire test_kalman) ou LS (sous-répertoire test_ls).

Les deux sous-dossiers test_kalman et test_ls contiennent chacun :

- un script matlab (gen_batch.m ou gen_batch_ls.m) pour générer automatiquement des fichiers de paramètres dans le dossier batch, en fonction du diamètre du télescope, du bruit de détecteur de l'ASO, et du gain pour LS ou fudge factor pour Kalman.
- un script matlab pour générer automatiquement les courbes de résultats (strehl et gain/fudge factor en fonction du bruit total (bruit détecteur + bruit de photons)) à partir du fichier de résultats resultats_scripts_kalman.dat ou résultats_scripts_ls.dat.
- un ou deux fichier de paramètres contenant les même caractéristiques (ASO, dimensionnement du télescope, turbulence, etc.), sauf pour le controlleur :
  - controlleur kalman_CPU pour le fichier test_kalman_zonal_CPU.par
  - controlleur kalman_GPU pour le fichier test_kalman_zonal_GPU.par
  - controlleur LS pour le fichier ls__08_001.6600_000000.600.par
- un script yorick pour exécuter tous les fichiers en batch
- un dossier batch, dans lequel sont générés tous le fichiers de paramètres lors de l'exécution du script matlab gen_batch ou gen_batch_ls

La Partie B : VIII) détaille comment fonctionne et comment utiliser yorick en batch.

II) Classe *sutra_controller_kalman*

La classe *sutra_controller_kalman* a été ajoutée à libsutra. Cette classe hérite de la classe *sutra_controller*.
En plus des attributs hérités (d_centroids, d_com, etc.), elle contient les attributs privés suivants :

- *core_sparse :* pointeur sur un objet de classe *kp_kalman_core_sparse.* Lors de la création de la classe, le pointeur est initialisé à NULL et le restera si les matrices *D_Mo, N_Act* et *PROJ* ne sont pas représentées en tant que matrices creuses.
- *core_full :* pointeur sur un objet de classe *kp_kalman_core_full.* Lors de la création de la classe, le pointeur est initialisé à NULL et le restera si les matrices *D_Mo, N_Act* et *PROJ* ne sont pas représentées en tant que matrices pleines.
- *isGPU* : booléen. Si *isGPU* vaut *true* alors les calculs seront effectués sur GPU par la création d'un objet de la classe *kp_kalman_core_<sparse/full>_GPU*. Si *isGPU* vaut *false* alors les calculs seront effectués sur CPU par la création d'un objet de la classe *kp_kalman_core_<sparse/full>_CPU*.
- *isSparse* : booléen. Si *isSparse* vaut *true* alors les calculs seront effectués avec des matrices *D_Mo, N_Act* et *PROJ* représentées sous forme de matrices creuses par la création d'un objet de la classe *kp_kalman_core_sparse_<CPU/GPU>*. Si *isSparse* vaut *false* alors les calculs seront effectués avec des matrices D_Mo, N_Act et PROJ représentées sous forme de matrices pleines par la création d'un objet de la classe *kp_kalman_core_full_<CPU/GPU>*.
- *isInit* : booléen. Permet de s'assurer que l'initialisation ne se fera qu'une seule fois au maximum, et que l'initialisation a été faite avant le calcul du gain de Kalman.
- *isGainSet* : booléen. Permet de s'assurer que la valeur de kW (y_controllers.gain) a été prise en compte avant le calcul du gain de Kalman.
- *gain* : float. Valeur de kW.

En plus des méthodes héritées, la classe *sutra_controller_kalman* contient les méthodes suivantes :

- *init_kalman* : initialise les attributs *isGPU*, *isZonal* et *isSparse* de classe et crée un objet kp_kalman_core_<sparse/full>_<CPU/GPU> en fonction de *isSparse* et *isGPU*.
- *calculate_gain* : calcule le gain de Kalman en appelant la méthode *calculate_gain* de l'attribut *core_<sparse/full>*.
- *comp_com :* calcule les tensions à partir des pentes en appelant la méthode *comp_com* de l'attribut *core_<sparse/full>*.

III) Modification des classes et fonctions existantes

Des modifications ont également été apportées à certaines fonctions et classes :

- Dans sutra_rtc::add_controller (libsutra/src.cpp/sutra_rtc.cpp), création du controlleur sutra_controller_kalman, si y_controllers.type vaut « kalman_CPU » ou « kalman_GPU », dans le fichier de paramètres.

- Dans rtc_init (yoga_ao/yorick/yoga_ao.i) :
  - copie de la valeur du paramètre y_controllers.gain, correspondant à kW, dans l'attribut gain de l'objet sutra_controller_kalman.
  - création des matrices D_Mo, N_Act et PROJ.
  - création de la matrice SigmaTur et des vecteurs atur et btur.
  - détermination de l'ordre du modèle AR à partir de btur.
  - choix de la base (zonale/modale) et de la représentation des matrices D_Mo, N_Act et PROJ (matrices creuses/pleines).
  - création de la matrice SigmaV à partir de la matrice SigmaTur, de la base considérée, de l'ordre AR et des vecteurs atur et btur.
  - initialisation du controlleur : création d'objet kp_kalman_core puis calcul du gain de Kalman.

Les modifications apportées sont résumées ci-dessous :

```
<<script_system, yoga_ao/yorick/ao_script.i>>
...
rtc_init
  └► <<rtc_init, yoga_ao/yorick/yoga_ao.i>>
     ...
     rtc_addcontrol
       └► <<Y_rtc_addcontrol, yoga_ao/yoga_ao.cpp>>
          rtc_handler->add_controller
            └► <<sutra_rtc::add_controller, libsutra/src.cpp/sutra_rtc.cpp>>
               Si controller de type kalman_CPU ou kalman_GPU {
                 creation du controller sutra_controller_kalman }

     Si controller de type kalman_CPU ou kalman_GPU{
       rtc_setgain(kW)
         └► <<Y_rtc_setgain, yoga_ao/yoga_ao.cpp>>
            Si controller de type kalman_CPU ou kalman_GPU {
              control->setgain(kW) }
               └► <<sutra_controller_kalman::set_gain,
                  libsutra/src.cpp/sutra_controller_kalman.cpp>>
                  gain = kW ;

       Si controller de type kalman_GPU alors isGPU=1, sinon isGPU=0.
       D_Mo = create_dmo(1,1) ; N_Act = create_nact(1); PROJ = Lusolve(N_Act);
       SigmaTur = create_sigmaTur(1) ; atur = array(...) ; btur = array(...) ;
       ordreAR = anyof(btur)+1 ; isZonal=1 ; isSparse=1 ;
       SigmaV = create_sigmav(SigmaTur, isZonal, ordreAR, atur, btur);
       rtc_initkalman(D_Mo, N_Act, PROJ, SigmaV, atur, btur, isZonal, isSparse, isGPU);
         └► <<Y_rtc_initkalman, yoga_ao/yoga_ao.cpp>>
            control->init_kalman(D_Mo, N_Act, PROJ, SigmaV, atur, btur, isZonal,
            isSparse, isGPU);
               └► <<sutra_controller_kalman::init_kalman,
                  libsutra/src.cpp/sutra_controller_kalman.cpp>>
                  Initialisation des attributs (booleens) de la classe :
                  is_zonal=isZonal ; is_sparse=isSparse ; is_GPU=isGPU ;
                  Si isSparse {
                    conversion de D_Mo, N_Act et PROJ en matrices creuses }
                  core=new kp_kalman_core_<sparse/full>_<CPU/GPU>(D_Mo,N_Act,PROJ,isZonal);
                     └► <<kp_kalman_core_*::kp_kalman_core_*,
                        lam/kalman_CPU_GPU/kp_kalman_core_*.cpp>>
                        Copie des matrices D_Mo, N_Act et PROJ dans la classe (soit sur CPU,
                        soit sur GPU).
                        Allocation des vecteurs de la classe qui seront utilises pour
                        calculer les tensions a partir des pentes.
            control->calculate_gain(bruit, SigmaV, atur, btur);
               └► <<sutra_controller_kalman::calculate_gain,
                  libsutra/src.cpp/sutra_controller_kalman.cpp>>
                  core->calculate_gain(bruit, gain, SigmaV, atur, btur) ;
                     └► <<kp_kalman_core_*::calculate_gain,
                        lam/kalman_CPU_GPU/kp_kalman_core_*.cpp>>
                        Calcul du gain de Kalman qui sera utilise pour calculer les tensions
                        a partir des pentes.
     }
...
for (cc=1;cc<=y_loop.niter;cc++) {
  ...
  rtc_docontrol
    └► <<Y_rtc_addcontrol, yoga_ao/yoga_ao.cpp>>
       rtc_handler->do_control
         └► <<sutra_rtc::do_control, libsutra/src.cpp/sutra_rtc.cpp>>
            this->d_control[ncntrl]->comp_com();
              └► <<sutra_controller_kalman::comp_com,
                 libsutra/src.cpp/sutra_controller_kalman.cpp>>
                 core->next_step(d_centroids, d_com);
                    └► <<kp_kalman_core_*::next_step,
                       Lam/kalman_CPU_GPU/ka_kalman_core_*.cpp>>
                       Calcul des tensions a partir des pentes
  ...
}
...
```

IV) Unités utilisées dans COMPASS

Variable	Nom dans COMPASS	Unité dans COMPASS	Nom dans classe kp_kalman_core*	Unité dans kp_kalman_core*
tensions appliquées aux actionneurs	d_com	V	U_k, U_km1, U_km2	V
pentes mesurées sur l'ASO	d_centroids	arcsec	Y_k	arcsec
matrice de covariance de la turbulence :	SigmaTur = create_sigmaTur(1)	rad²	Non définie	Non définie
diagonale de la matrice de covariance du bruit de mesure :	SigmaW = noise_cov(1)	arcsec²	bruit	arcsec²
matrice de l'ASO	D_Mo	arcsec/micron	D_Mo	arccsec/micron
matrice d'influence	N_Act	rad/V	N_Act	rad/V
Inverse de la matrice d'influence	PROJ	V/rad	PROJ	V/rad
matrice de covariance du bruit de modèle	SigmaV= create_sigmav (sigma_tur, ...)	même unité que sigma_tur	SigmaV	micron²

V) Conversions effectuées

Dans la fonction create_dmo (fichier yoga_ao/yorick/yoga_rtc.i), la variable *coeff=pixsize*2*pi/lambda* permet de passer d'obtenir une matrice *D_Mo* en arcsec/micron à partir de celle en px/rad : *D_Mo [arcsec/micron] = D_Mo [px/rad] * coeff*. La matrice retournée par *D_Mo* est donc en arcsec/micron.

Dans rtc_init (yoga_ao/yorick/yoga_ao.i), on convertit la matrice retournée par *create_sigmaTur* (en rad²) en une matrice en micron² : *SigmaTur = create_sigmaTur(1) * (lambda/2/pi)^2*;

Dans l'expression de l'innovation, (Y_k - Y_kskm1) a été remplacé par (Y_k + Y_kskm1)

VI) Utilisation du controlleur Kalman dans COMPASS

Cette partie décrit la liste des paramètres modifiables dans COMPASS concernant le controlleur Kalman.

1) Sélection du controlleur Kalman

Pour choisir le controller Kalman, il faut modifier le fichier de paramètres (fichier .par).

C'est dans la partie *controllers inits*, qu'il faut paramétrer le controller Kalman.
Voici un exemple de paramétrage d'un controller Kalman version GPU, utilisant un ASO et un MD :

```
/*controllers inits*/
ncontrollers = 1;
y_controllers = array(controller_struct(),ncontrollers);
y_controllers(1).type = "kalman_GPU";
y_controllers(1).nwfs = &[1];
y_controllers(1).ndm  = &[1];
y_controllers(1).gain = 320;
```

Le paramètre `y_controllers(1).gain` correspond au fudge factor kW (cf partie Partie B : VI) 2).

Le filtre de Kalman tel qu'il a été conçu nécessite un unique miroir PZT, donc pas de miroir TT). **Il faut donc définir un seul miroir, de type «pzt», dans la partie *dm inits* du fichier .par :**

```
/*dm inits*/
ndm = 1;
y_dm                = array(dm_struct(),ndm);
y_dm(1).type        = "pzt";
y_dm(1).nact        = y_wfs(1).nxsub+1;
y_dm(1).alt         = 0.;
y_dm(1).thresh      = 0.44;
y_dm(1).coupling    = 0.3;
y_dm(1).unitpervolt = 1.0;
y_dm(1).push4imat   = 1.0f;
```

2) Valeur du fudge factor kW

Dans le cas du controleur Kalman, on introduit un paramètre, nommé fudge ***factor***, noté ***kW*** ou ***k_W***, permettant de compenser les erreurs de modèle. Cette valeur de kW correspond à la valeur du champ `y_controllers(1).gain` (où `y_controllers(1)` correspond au controlleur Kalman). Ce gain sera ensuite stockée en tant que float, dans l'attribut ***gain***, lors de l'appel à la fonction *rtc_setgain*, appelée par *rtc_init* (fichier).

3) Sélection du format de matrices (creuses/pleines)

Il est possible de choisir d'utiliser le format de matrices pleines ou de matrices creuses pour les matrices D_Mo, N_Act et PROJ.

Le format utilisé est pris en compte dans le fichier yoga_ao/yorick/yoga_ao.i, dans la fonction ***rtc_init***. C'est la variable ***isSparse*** qui détermine le format :

- si **i*sSparse = 1*** alors le format de matrices creuses sera utilisé
- si ***isSparse = 0*** alors le format de matrices pleines sera utilisé

4) Sélection de la base (zonale/modale)

Il est possible de choisir d'utiliser une base zonale (base des actionneurs) ou une base modale (base des Zernike). Cependant dans le code actuel permet de calculer les matrices D_Mo, N_Act et PROJ uniquement en base zonale.

Le base utilisée est prise en compte dans le fichier yoga_ao/yorick/yoga_ao.i, dans la fonction ***rtc_init***. C'est la variable ***isZonal*** qui détermine la base :

- si **i*sZonal = 1*** alors la base zonale sera utilisée
- si ***isZonal = 0*** alors la base modale sera utilisée

5) Sélection de la version de Kalman (CPU/GPU)

Il est possible de choisir de faire tourner le filtre de Kalman, soit sur CPU, soit sur GPU.

La version utilisée est prise en compte dans le fichier de paramètre. C'est la valeur du champ ***y_controllers(1).type*** qui détermine la version du controlleur (où ***y_controllers(1)*** correspond au controlleur Kalman) :

- si ***y_controllers(1).type = "kalman_CPU"*** alors la version CPU sera utilisée
- si ***y_controllers(1).type = "kalman_GPU"*** alors la version GPU sera utilisée

VII) Compilation

Si la variable d'environnement *COMPILATION_LAM* est définie et non vide, alors le Makefile dans trunk appellera le Makefile dans trunk/lam pour compiler les bibliothèques statiques nécessaires pour l'utilisation du filtre de Kalman : *lib_kp_classes.a* dans trunk/lam/kp_classes et *lib_kalman.a* dans trunk/lam/kalman_CPU_GPU. Ces bibliothèques seront ajoutées lors de la compilation de libsutra, car elle sont nécessaires pour la classe *sutra_controller_kalman*.

Pour pouvoir utiliser le filtre de Kalman en version GPU, les bibliothèques suivantes sont nécessaires :

- cuda, cudart, cuBLAS, cuSPARSE, Thrust (CUDA API)
- Magma

Si la variable d'environnement *COMPILATION_LAM* n'est pas définie ou est définie mais vide alors les bibliothèques statiques ne seront pas générées (donc pas incluses lors de la compilation de libsutra). Il ne sera donc pas possible d'utiliser le controller kalman dans ce cas.

Lors de la compilation, si la variable d'environnement *COMPILATION_LAM* contient le mot «double» alors le filtre de Kalman sera compilé en double précison. Si *COMPILATION_LAM* ne contient pas ce mot et est non vide, alors le filtre de Kalman sera compilé en simple précision, sauf le calcul du gain de Kalman, qui nécessite la double précision. Si la variable *COMPILATION_LAM* est vide ou non définie alors la partie Kalman ne sera pas compilée.

Lors de la compilation, si la variable d'environnement *COMPILATION_LAM* contient le mot «standalone» alors le filtre de Kalman sera compilé pour la version standalone. Si *COMPILATION_LAM* ne contient pas ce mot et est non vide, alors le filtre de Kalman sera compilé pour COMPASS. Si la variable *COMPILATION_LAM* est vide ou non définie alors la partie Kalman ne sera pas compilée.

La seule différence algorithmique du filtre de Kalman entre les versions standalone et pour COMPASS est le signe intervenant dans le calcul de l'innovation, dans la méthode *next_step* des classes *kp_kalman_core_** :

- version standalone : $innovation = Y\_k - Y\_kskm1$
- version pour COMPASS : $innovation = Y\_k + Y\_kskm1$

VIII) Utilisation du controlleur Kalman dans COMPASS

Le répertoire trunk/lam/test_compass permet d'effectuer des simulations en batch pour les controlleurs Kalman et LS.

Comme le principe est identique pour les deux controlleurs, seul le cas du controlleur Kalman est détaillé dans la suite.

Il est possible d'utiliser le mode batch de deux manières différentes : soit en prenant en compte tous les fichiers présent dans le répertoire batch (ce mode est appelé **batch implicite**), soit en détaillant explicitement le nom de tous les fichiers de paramètres, que l'on souhaite prendre en compte (ce mode est nommé **batch explicite**).

1) Utilisation du batch explicite

Pour utiliser le mode batch explicite avec le controlleur Kalman, il suffit de se placer dans le répertoire *lam/test_compass/test_kalman* et d'exécuter la commande suivante :

yorick -i ao_scripts_debug_batch_kalman.i fichier1.par [fichier2.par] [...] [fichierN.par]

Lors de l'exécution de ce script yorick, tous les fichiers de paramètres renseignés seront exécutés un par un. Les résultats obtenus à l'issue de l'exécution de chacun de ces fichers seront ajoutés au fichier *resultats_scripts_kalman.dat* (le fichier sera créé s'il n'existe pas).

Ces résultats sont ajoutés au fichier *resultats_scripts_kalman.dat* sous la forme d'une ligne contenant, dans l'ordre, ces différentes valeurs spéparées par un espace :

- diamètre du télescope en mètre (même valeur que y_tel.diam)
- bruit de détecteur en électron RMS (même valeur que y_wfs.noise)
- fudge factor pour Kalman ou gain pour LS (même valeur que y_controllers.gain)
- rapport de Strehl (même valeur que strehllp(0) à la dernière itération)

2) Utilisation du batch implicite

Pour utiliser le mode batch implicite avec le controlleur Kalman, il suffit de se placer dans le répertoire *lam/test_compass/test_kalman* et d'exécuter la commande suivante :

yorick -i ao_scripts_debug_batch_kalman.i

Au début de l'exécution de cette commande, le fichier *liste_fichier_batch.txt* sera créé. Il contiendra tous les noms de fichiers du répertoire *batch* (un nom de fichier par ligne), qui sont censés être des fichiers de paramètres (.par).

Ensuite, le script bouclera sur chaque fichier présent dans le répertoire *batch* (dont le nom figure dans *liste_fichier_batch.txt*) pour exécuter un à un l'ensemble des fichiers.
Les résultats obtenus à l'issue de l'exécution de chacun de ces fichers seront ajoutés au fichier *resultats_scripts_kalman.dat* (le fichier sera créé s'il n'existe pas), sous la même forme que dans le cas du batch explicite.

Le srcript matlab *gen_batch.m* (ou *gen_batch_ls.m*) permet de générer automatiquement des fichiers de paramètres dans le répertoire *batch*, en fonction du bruit de détecteur, du diamètre du télescope et du fudge factor (ou gain pour LS).

Le script matlab *res_batch.m* permet d'afficher les courbes de strehl et de fudge factor (ou gain pour LS) en fonction du bruit total sur l'ASO (bruit de détecteur + bruit de photons)

ATTENTION, dans ces 2 scripts matlab, la correspondance entre le bruit de détecteur (*bruit* ou *bruit_détecteur_erms*) et le bruit total en px² (*bruit_gamma_px2*) n'est valable que pour une magnitude de l'étoile guide (*y_wfs.gsmag*) de 5.

IX) Résultats obtenus

Différentes simulations en mode batch implicite ont été effectuées pour obtenir les résultats de rapport de strehl et de fudge factor (ou gain pour LS) en fonction du bruit total, en faisant varier le bruit de détecteur. Comme le bruit de photons augmente avec le diamètre, plus le diamètre est élevé, plus le premier point de la courbe que l'on peut obtenir correspond à un bruit plus elevé, car même avec un bruit de détecteur nul, le bruit de photons est toujours là.
Cela explique pourquoi il y a moins de points sur les courbes de 40m que sur celles de 8m.

Dans le cas du LS, la valeur de max_cond est de 150.0 pour le 8m et 250.0 pour le 40m.

Les tests pour les LS ont été effectuées avec une valeur de gain variant de 0,05 à 0,65, par pas de 0,05 pour chaque valeur de bruit de chaque diamètre. Seuls les résultats du gain donnant le meilleur rapport de strehl sont pris en compte dans les figures suivantes.

Les tests pour Kalman ont été effectué en deux temps. Dans un premier temps, pour chaque valeur de bruit de chaque diamètre, on fait varier la valeur du fudge factor kW, suivant une suite géométrique de premier terme 1 et de raison 1,8.
Dans un second temps, pour chaque valeur de bruit de chaque diamètre, on affine la valeur du kW autour du meilleur kW de la première partie.
Seuls les résultats du kW donnant le meilleur rapport de strehl sont pris en compte dans les figures suivantes.

Ce test en deux temps pour Kalman a été effectuée lors des toutes premières simulations en batch, car ensuite la plage des kW est connue pour chaque valeur de bruit.

Tous les résultats décrits dans la suite, ont été effectués avec les paramètres suivants constants et identique pour Kalman et Ls afin de pouvoir comparer les deux controlleurs :

```
y_geom.zenithangle = 0.;
y_tel.cobs = 0.f;
y_atmos.r0           = 0.1249;
y_atmos.nscreens     = 3;
y_atmos.frac         = &([0.5,0.17,0.33]);
y_atmos.alt          = &([0.0,2500.,4000.]);
y_atmos.windspeed    = &([7.5,12.5,15]);.
y_atmos.winddir      = &([0,90,0]);
y_atmos.L0           = &([25,25,25]);
y_target.ntargets = 1;
y_target.xpos     = &([0.0f]);
y_target.ypos     = &([0.0f]);
y_target.lambda   = &([1.654]);
y_target.mag      = &([10.]);
y_loop.niter  = 2000;
y_loop.ittime = 1/500.0f;
y_wfs                   = array(wfs_struct(),1);
y_wfs(1:).type          = "sh";
y_wfs(1:).npix          = 14;
y_wfs(1:).pixsize       = 0.3412;
y_wfs(1:).fracsub       = 0.5;
y_wfs(1:).xpos          = 0.0;
y_wfs(1:).ypos          = 0.0;
y_wfs(1:).lambda        = 1.654;
y_wfs(1:).gsmag         = 5.;
y_wfs(1:).optthroughput = 0.5;
y_wfs(1:).zerop         = 1.e11;
ndm = 1;
```

```
y_dm                  = array(dm_struct(),ndm);
y_dm(1).type          = "pzt";
y_dm(1).nact          = y_wfs(1).nxsub+1;
y_dm(1).alt           = 0.;
y_dm(1).thresh        = 0.44;
y_dm(1).coupling      = 0.3;.
y_dm(1).unitpervolt   = 1.0;.
y_dm(1).push4imat     = 1.0f;
ncentroiders = 1;
y_centroiders  = array(centroider_struct(),ncentroiders);
y_centroiders.nwfs      = 1;
y_centroiders(1).type   = "cog";
ncontrollers = 1;
y_controllers  = array(controller_struct(),ncontrollers);
y_rtc.nwfs =1;
y_rtc.centroiders = &(y_centroiders);
y_rtc.controllers  = &(y_controllers)
```

1) Résultats obtenus en ASO géométrique

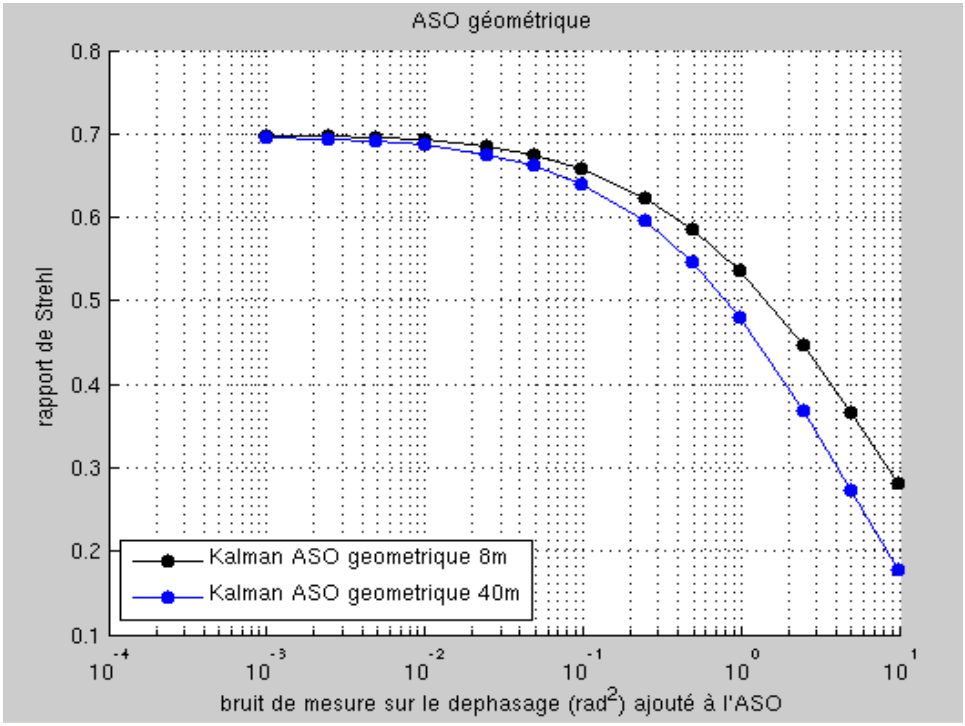
Pour obtenir des résultats avec l'ASO géométrique, il a été necessaire d'apporter quelques modifications à COMPASS. Ces modifications n'ont pas été « commitées », car elles impliquent beaucoup de changement dans l'utilisation de COMPASS et une non-compatibilité, à moins d'effectuer encore d'autres changements. Voici en quoi consiste ces modifications :

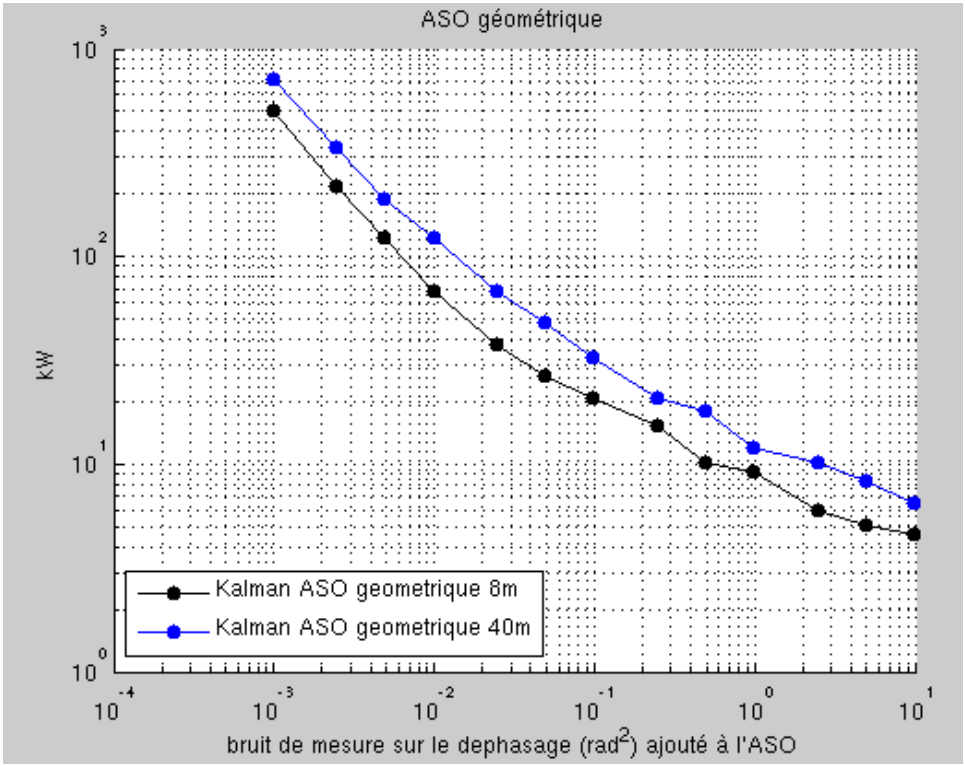
- Pour LS :
  - Appel à la fonction *rtc_docentroids_geom* au lieu de *rtc_docentroids*
  - Dans les fichiers de paramètre, le paramètre *y_wfs.noise* ne correspond plus au bruit de détecteur en électron RMS, mais au bruit (variance) en arcsec².
  - Ajout de l'attribut *rac_bruit_alpha_asec* et de la méthode *init_ls_bruit* dans la classe *sutra_controller_ls*, correspondant à l'écart-type du bruit en arcsec.
  - Ajout de la fonction yorick *rtc_init_ls_bruit* (appelée dans la fonction *rtc_init* du fichier *yoga_ao.i*) et du wrapper yorick/C *Y_rtc_initlsbruit*, permettant d'initialiser l'attribut *rac_bruit_alpha_asec* de *sutra_controller_ls* à partir de sa méthode *init_ls_bruit*.

    La variance du bruit *y_wfs.noise* est transmise à la fonction *rtc_init_ls_bruit* en tant qu'argument, permettant d'initialiser l'attribut *rac_bruit_alpha_asec* comme la racine carée de cet argument.
  - Un vecteur de bruit gaussien centré d'écart-type *rac_bruit_alpha_asec* est ajouté aux pentes au début de la méthode *sutra_controller_ls::comp_com*,

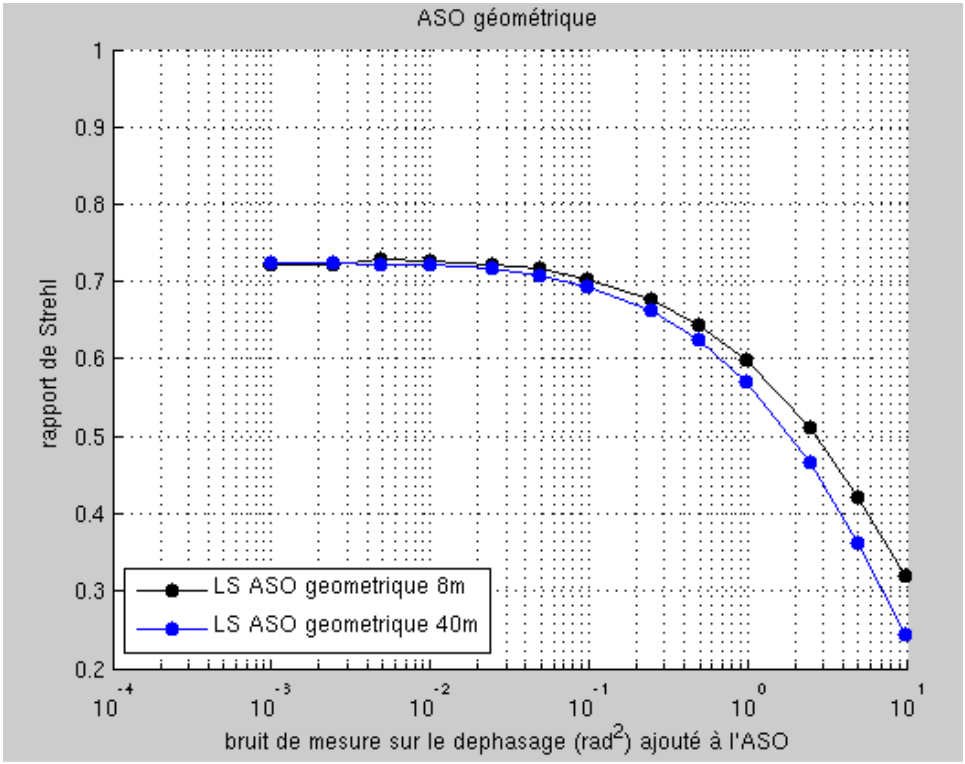
- Pour Kalman :
  - Appel à la fonction *rtc_docentroids_geom* au lieu de *rtc_docentroids*
  - Ajout de l'attribut *rac_bruit_alpha_asec* dans la classe *sutra_controller_kalman.*
  - La fonction rtc_initkalman (appelée dans la fonction *rtc_init* du fichier *yoga_ao.i*) ne prend plus comme argument la moyenne de la diagonale de la matrice de covariance du bruit de mesure en arcsec² (*avg(noise_cov(1)*), qui n'a plus de sens), mais simplement la variance du bruit *y_wfs(1).noise.* Cela permet d'initialiser l'attribut *rac_bruit_alpha_asec* comme la racine carrée de cet argument.
  - Dans la méthode *sutra_controller_kalman::comp_com*, un vecteur de bruit gaussien centré d'écart-type *(bruit)* est ajouté aux pentes.

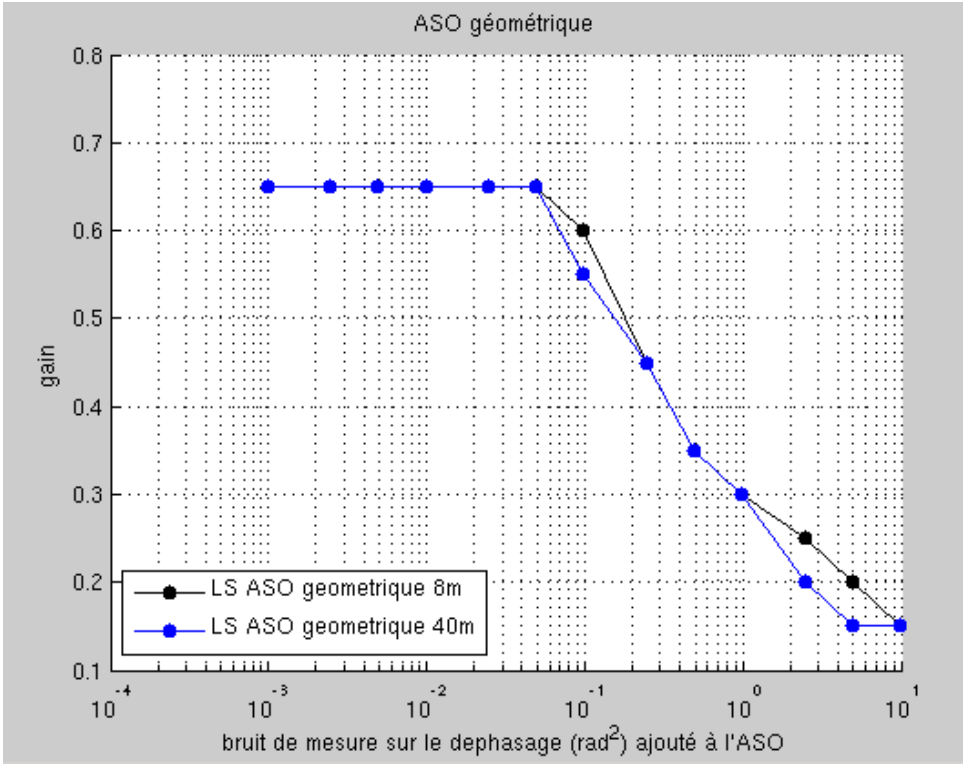
a) Résultats du Kalman : rapport de strehl et fudge factor en fonction du bruit total



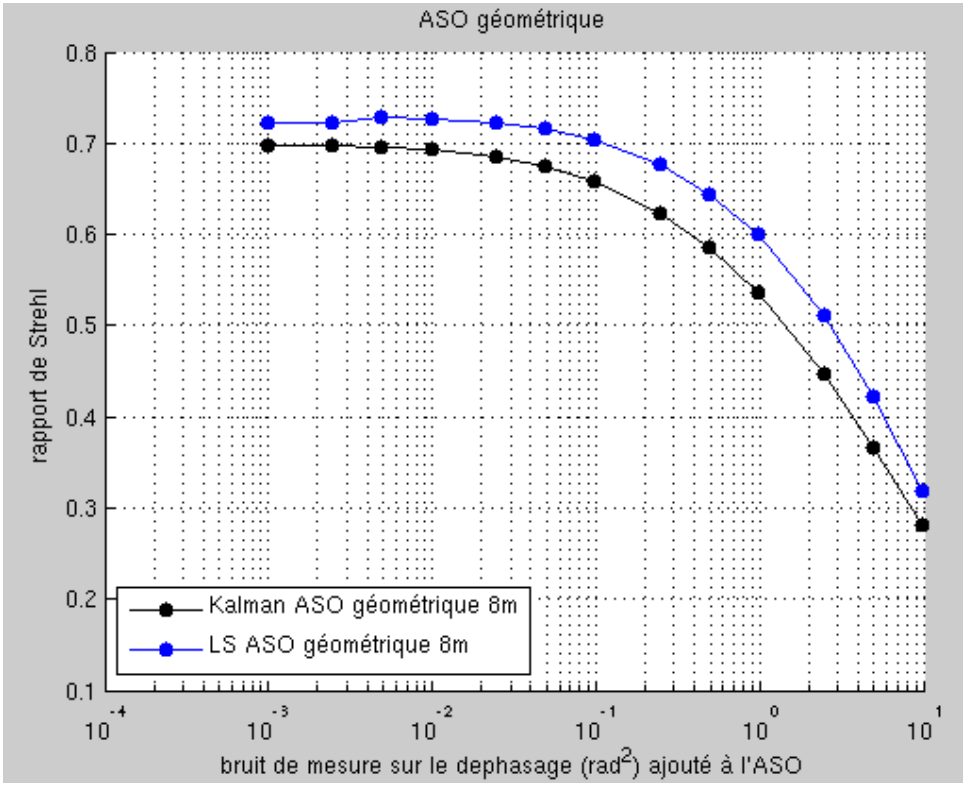


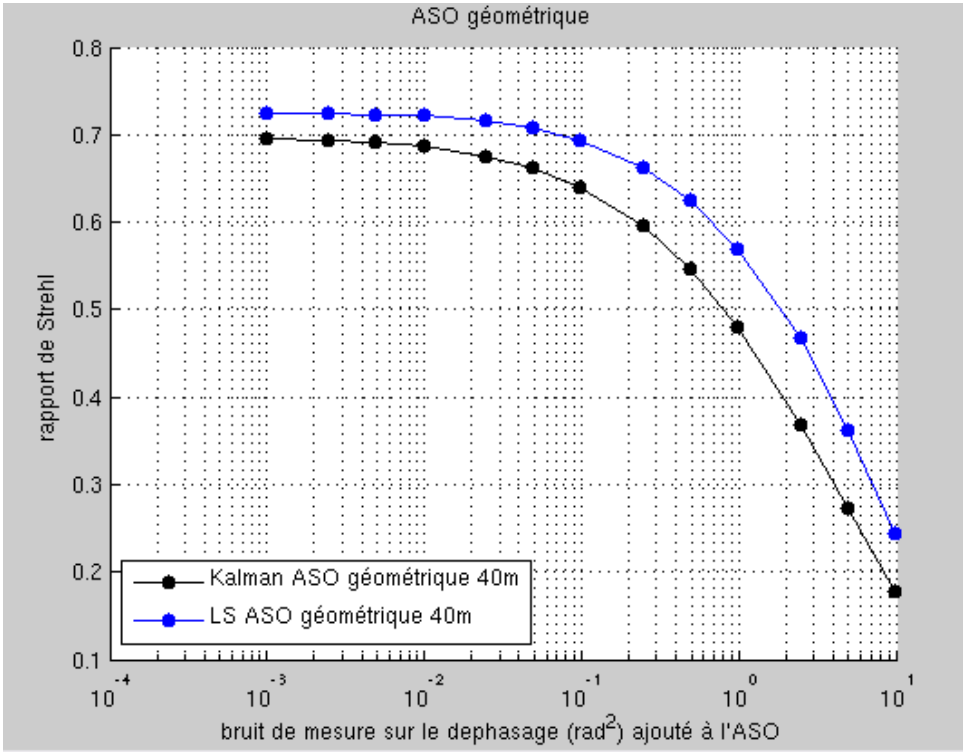
b) Résultats du LS : rapport de strehl et gain en fonction du bruit total





c) Comparaison Kalman-LS

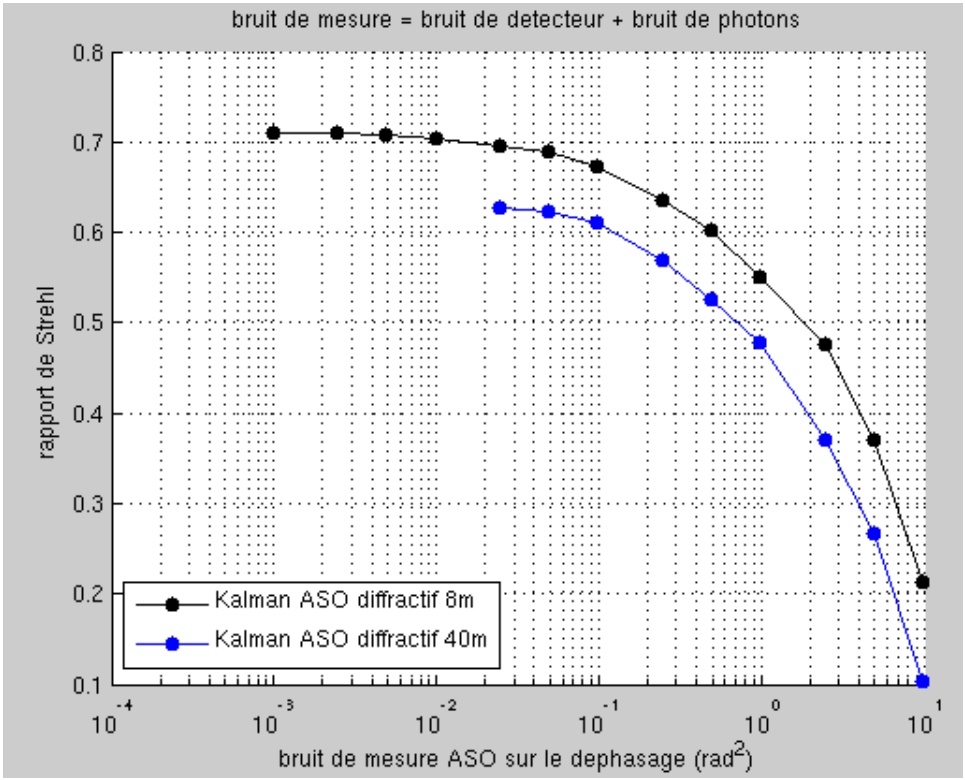


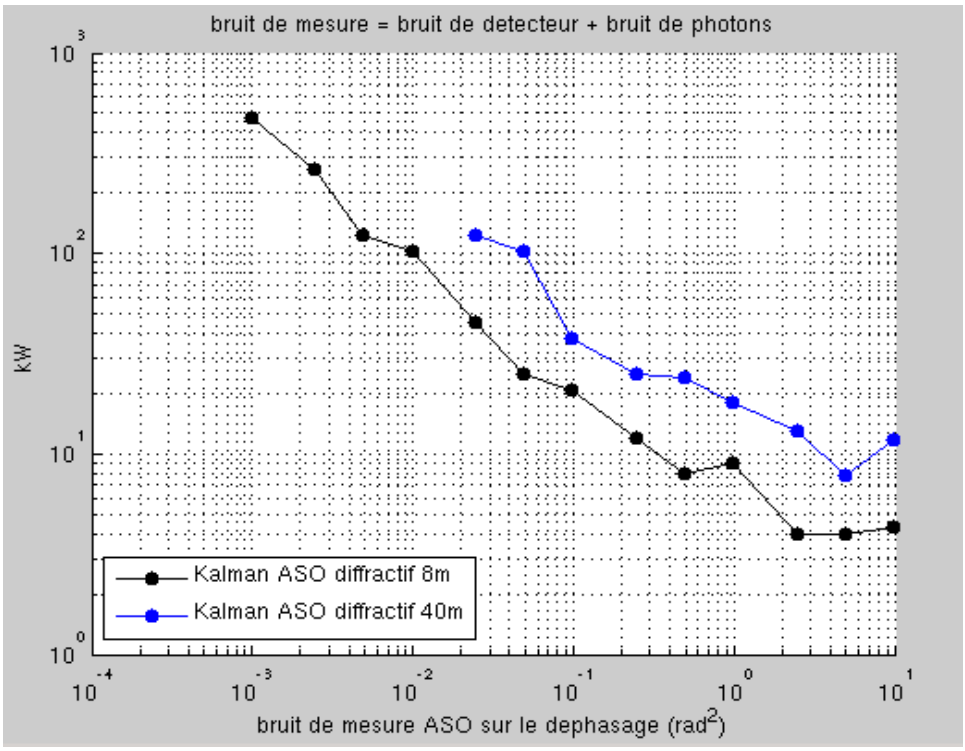


2) Résultats obtenus en ASO diffractif

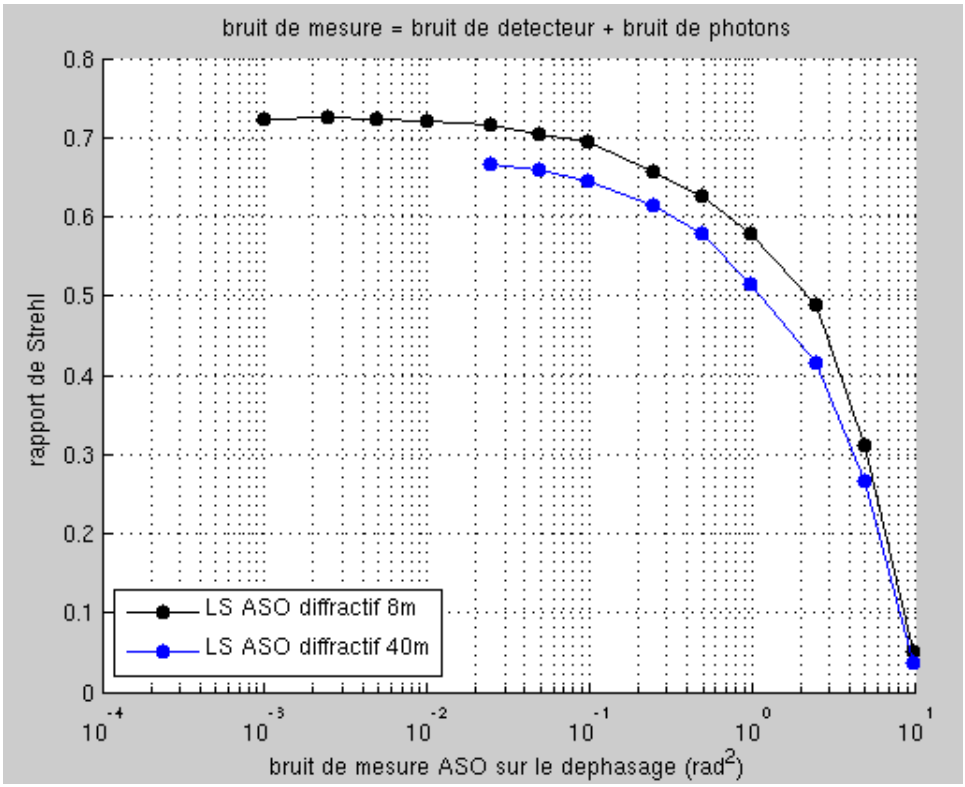
Ces résultats ont été obtenus avec la version classique (non modifiée) de COMPASS, en utilisant un ASO_diffractif

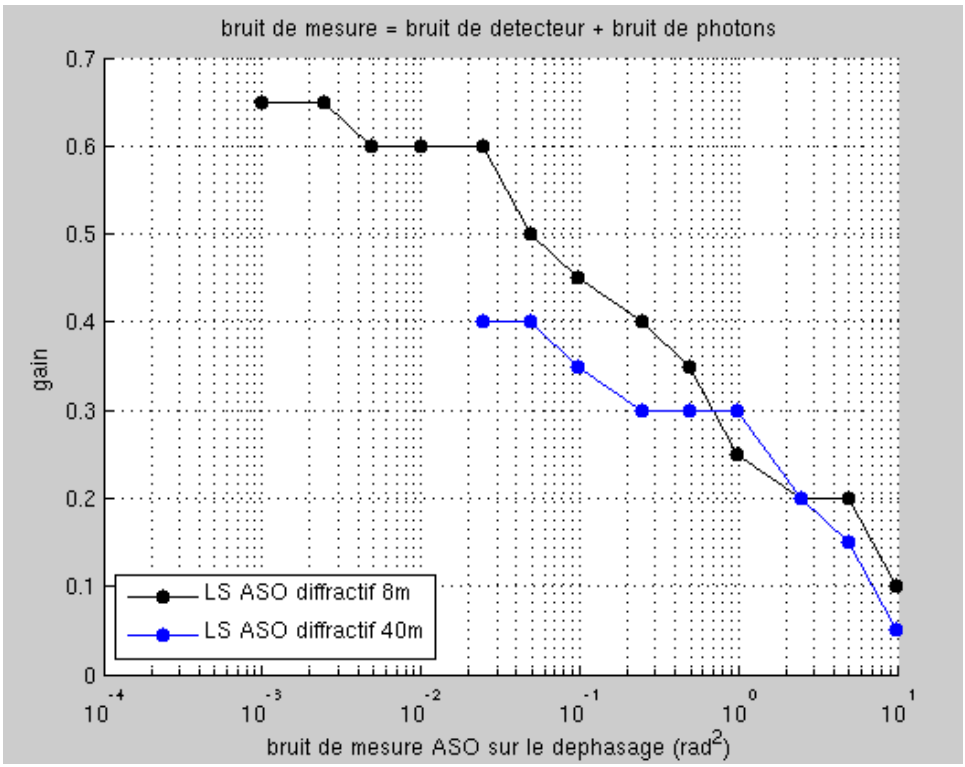
a) Résultats du Kalman : rapport de strehl et fudge factor en fonction du bruit total



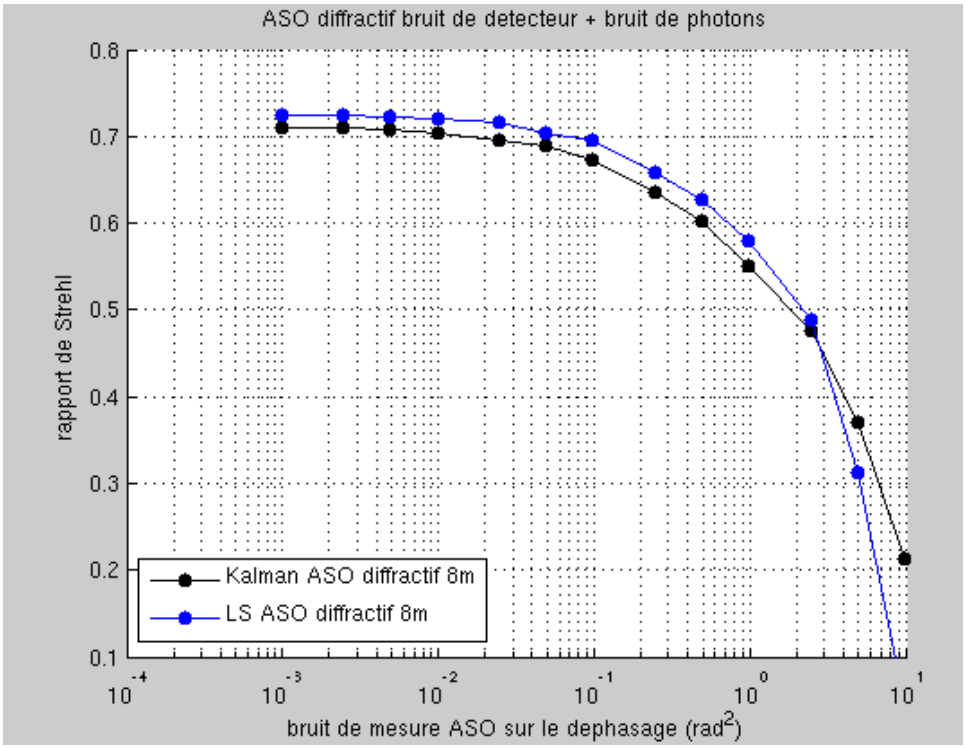


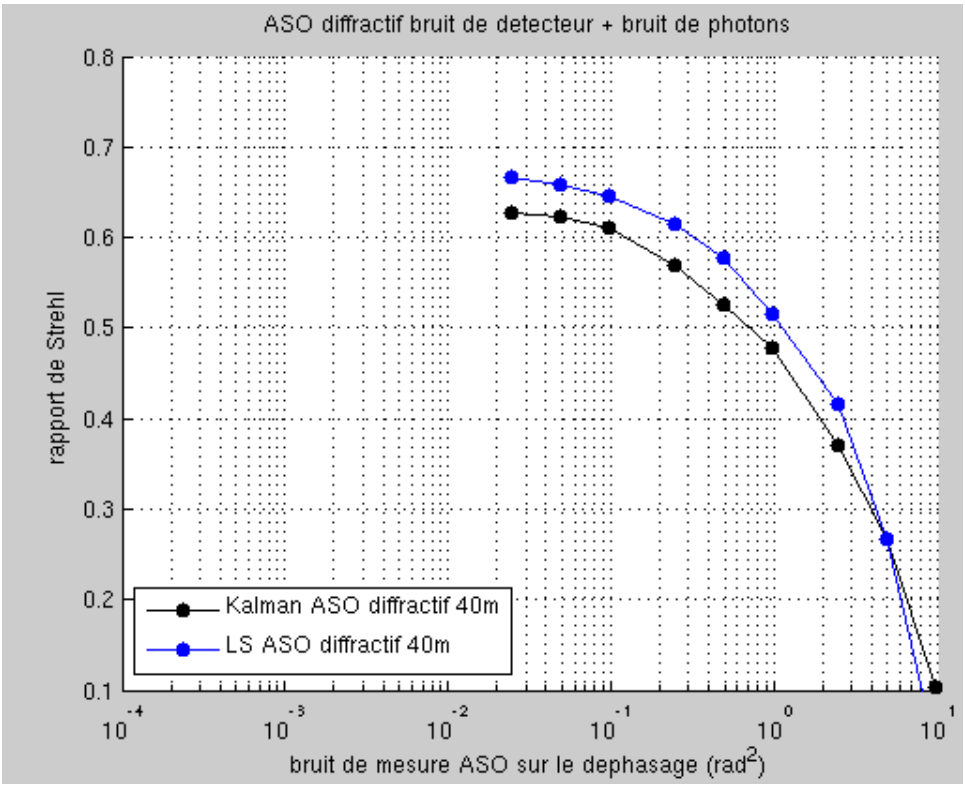
b) Résultats du LS : rapport de strehl et gain en fonction du bruit total





c) Comparaison Kalman-LS





Même si les courbes du Kalman ont la même que celles du LS, ces courbes révèlent quelques anomalies :

- On remarque, que dans le cas du controller Kalman, l'ASO diffractif fait parfois mieux que l'ASO géométrique.

- D'autre part, on s'attendait à ce que Kalman ferait mieux que le LS, mais on s'aperçoit que Kalman est toujours légèrement inférieur au LS.

- Les valeurs de kW sont beaucoup plus grandes que celles qu'on est censé avoir.