Qualité et reproductibilité numérique

TP sur la sommation flottante

Cet exercice permet la mise en pratique et l'illustration des différents points vus en cours concernant l'arithmétique élémentaire de la norme IEEE-754, la perte de précision lors d'un enchaînement de calculs, la perte de propriétés arithmétiques (e.g., associativité de l'addition) et l'influence de l'ordre des opérations arithmétiques sur la précision de la solution calculée, l'influence du conditionnement du problème sur la précision de la solution calculée avec un algorithme inverse stable.

On s'intéresse au calcul de la somme s de n nombres flottants x_i, $s = \sum_{i=0}^{n-1} x_i$. Il s'agit d'observer la qualité et la reproductibilité numériques de différents algorithmes qui calculent cette somme. Qualité et reproductibilité dépendent (au moins) du conditionnement des jeux de données $(x_i)_i$, de leurs taille n. La performance de certains algorithmes dépend aussi de l'exposant des entrées $(x_i)_i$.

1 Démarche générale

1.1 Les données.

Il y a plusieurs types d'entrées x_i à considérer.

- données aléatoires de même signe, par exemple selon une distribution uniforme $U(0, 2)$
- données aléatoires de signe quelconques, par exemple selon une distribution uniforme $U(-1, 1)$
- données de *range* d'exposants variables
- données de conditionnement et de taille variables

Pour la partie 1, des jeux de données de ce dernier type sont sont proposés dans des fichiers (texte) disponibles dans l'archive. Vous pourrez développer les autres cas.
Pour la partie 2, des générateurs pour les 2 derniers types sont fournis dans l'archive et produisent les données sous form de fichiers binaires.

1.1.1 Fichiers de données avec dépendance au conditionnement

Ces fichiers définissent les valeurs de `n` (`int`), des opérandes `x[i]`, `i=0`, `n-1` (`binary64`) et le conditionnement de leur somme `C` (`binary64`) selon la structure suivante.

- Ligne 1 : `n`, `C` ;
- lignes 2 à $n + 1$: `x[i]`
- ligne n+2 : `S` arrondi `binary64` de la somme exacte s

Ces fichiers sont organisés comme suit. Le répertoire `data` contient 44 jeux de données pour 11 valeurs de conditionnements variant de 10^3 à 10^{32}. Quatre jeux de données sont disponibles pour chaque ordre de grandeur des conditionnements. Les noms de fichiers explicitent la taille (ici fixée à 100) et le conditionnement.

1.2 Les algorithmes.

Les algorithmes de sommation sont nombreux !
Dans la partie 1, vous choisirez et coderez ceux qui vous intéressent parmi les 4 types suivants.
Dans la partie 2, des implémentations C sont fournies dans l'archive.

1.2.1 Variations autour de l'algorithme naif

La somme est calculée par accumulation. Les algorithmes suivants ne diffèrent que par l'ordre de l'accumulation. Ce qui donne déjà un avant-goût sur la (non) reproductibilité des algorithmes parallèles.

A1. dans l'ordre croissant ou décroissant des indices : $((x_0 + x_1) + x_2) \cdots$,

A2. dans un ordre aléatoire :des indices obtenu par *shuffle* [1]

1. Un algorithme de shuffle est par exemple : $i = 0$; choisir j dans $[1, n - 1]$, permuter $x[i]$ et $x[j]$, incrémenter i et recommencer jusqu'à $i = n - 2$.

A3. opérandes positifs puis négatifs (ou l'inverse) dans l'ordre croissant (ou l'inverse) des indices,

A4. somme partielle S_+ des opérandes positifs, puis somme partielle S_- des opérandes négatifs chacune dans l'ordre croissant des indices, et somme de S_+ et S_-,

A5. ordre croissant des valeurs des opérandes,

A6. ordre décroissant des valeurs des opérandes,

A7. ordre croissant des valeurs absolues des opérandes,

A8. ordre décroissant des valeurs absolues des opérandes,

A9. addition récursive par paire (*pairwise*) :
$((x_0 + x_1) + (x_2 + x_3)) + ((x_4 + x_5) + (x_6 + x_7))$ pour $n = 8$ par exemple.

A10. supprimer les deux opérandes de plus plus petite valeur absolue, les additionner, ajouter cette somme comme un nouvel opérande et recommencer (addition des deux plus petites valeurs absolues, ...) jusqu'à ce qu'il n'y ait plus d'opérande à additionner.

1.2.2 Algorithmes à précision fixée

On a vu en cours :

— la somme compensée de Kahan

— celle de Pichat et al. : Sum2

— la doublement compensée de Priest

— l'accumulation en double-double

1.2.3 Algorithmes à précision arbitraire dépendant du conditionnement

On a vu en cours :

— SumK de Ogita-Rump-Oishi

1.2.4 Algorithmes fidèles ou correctement arrondis

On a vu en cours :

— les découpages AccSum et FastAccSum de Rump

— la distillation iFastSum

— les rangements par exposant HybridSum et OneLineExact de Zhu-Hayes

1.2.5 Algorithmes reproductibles

On a vu en cours :

— les découpages ReprodSum, FastReprodSum et OneReduction de Demmel-Nguyen

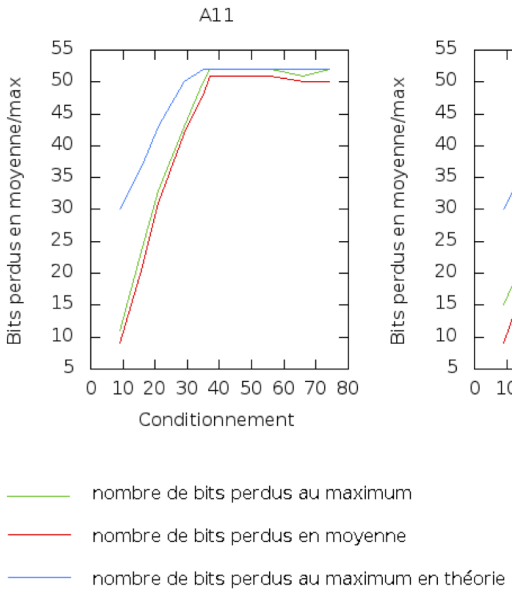
1.3 Les mesures.

L'erreur relative entre la solution calculée $\widehat{s}$ et la somme exacte s mesure la précision de $\widehat{s}$. La somme exacte s est inconnue en pratique. Nous en obtenons une valeur approchée S grâce à un calcul en précision suffisamment élévée pour que S permette de mesurer la précision de $\widehat{s}$.

2 Sur votre machine et avec votre langage préféré

1. Comparer la précision atteinte par ces différents algorithmes et l'effet du conditionnement sur cette précision. On procède en deux temps.
   (a) A l'aide des différents jeux de données fournis dans `data_txt`, s'assurer pour chaque algorithme que la précision évolue en fonction du nombre de conditionnement. Qu'en conclure.
   (b) Générer deux fichiers `moy.data` et `max.data` qui contiennent respectivement la moyenne et le maximum de cette perte de précision pour chaque algorithme et chaque ordre de grandeur du conditionnement.
   (c) Ecrire un fichier de commandes `gnuplot`, `matplotlib`, ..., pour tracer les deux graphiques suivants à partir des fichiers `moy.data` et `max.data`.
      i. L'erreur relative moyenne (ordonnée) comme fonction du conditionnement (abscisse),
      ii. L'erreur relative maximum comme fonction du conditionnement.

   Le conditionnement est exprimé sous la forme d'une puissance de 2. Une échelle logarithmique est adaptée à son affichage. L'erreur relative est mesurée en nombre de bits significatifs [2]. Chaque graphique regroupera les résultats de l'ensemble des algorithmes. Commenter les résultats ainsi obtenus.
2. Majorations
   (a) Rappeler la majoration *a priori* de l'erreur dans la somme calculée. Compléter les tracés précédents avec l'affichage de cette borne. Commenter les résultats ainsi obtenus.
   (b) Rappeler la majoration dynamique de l'erreur dans la somme calculée. Compléter les tracés précédents avec l'affichage de cette borne.
   (c) (Plus tard). Regarder l'influence de la taille n des données sur la finesse de ces majorations.



3 Sur le serveur et en C.

Nous détaillons ici l'analyse et les résultats (de précision et de performance) de huit algorithmes de sommation présentés au tableau 3.1. La démarche peut être étendue aux algorithmes de votre choix.

3.1 Les paramètres et leurs valeurs

La précision (et la performance) des algorithmes de sommation dépend des paramètres suivants.

— la longueur de la somme ou du vecteur des entrées x_i, soit n pour $\sum_{i=1}^{n} x_i$;

— le nombre de conditionnement de la somme, soit $cond = \sum_i |x_i| / |\sum_i x_i|$;

2. En binary64, NbBitSig = $53 - \log_2(|S - \widehat{s}|/|S|)$

— l'étendue des exposants (binaires) des valeurs à sommer, soit $\delta = e_{max} - e_{min}$ où $e_{max} = \max_e\{2^e \leq |x_i| < 2^{e+1}, i = 1, \cdots, n\}$ et $e_{min} = \min_e\{2^e \leq |x_i| < 2^{e+1}, i = 1, \cdots, n\}$. En pratique, δ varie dans l'intervalle $[0, 2046]$ pour les binary64 de l'IEEE-754.

Algorithme	Paramètres	Précision
Sum	n	Sommation classique
Sum2, DDSum	n, $cond$	Sommation (deux fois plus) précise
AccSum, FastAccSum, iFastSum	n, $cond$	Sommation fidèle
HybridSum, OnLineExact	n, $cond$, δ	Sommation fidèle

TABLE 1 – Paramètres des algorithmes de sommation. n : longueur de la somme, $cond$: nombre de conditionnement, δ : étendue de l'exposant

Les tableaux suivants donnent les valeurs des paramètres testés. Pour notre étude, les paramètres principaux sont n et $cond$. En effet l'étendue des exposants δ est fortement liée aux valeurs de $cond$ choisies : δ est en particulier une fonction croissante de $cond$ à n fixé.
Nous reviendrons plus tard sur l'influence particulière sur la performance de certains algorithmes de la distribution de ces exposants à n et δ fixés.

Paramètres	valeurs testées
n	$10^3, 10^4, 10^5, 10^6$
$cond$	$[10^8, 10^{40}] \approx [\mathbf{u}^{-1/2}, \mathbf{u}^{-2.5}]$
δ	$10, 100, 500, 1000, 1500, 2000$

TABLE 2 – Ensemble des valeurs de paramètres testés.

3.2 Préalable : génération des exécutables

Si il est utilisé seul, le compilateur gcc par défaut ne demande aucune modification particulière.

3.2.1 Paramétrages

Il s'agit d'initialiser les paramètres des algorithmes testés, soit ici les valeurs de n, $cond$ et δ. Certains grandes longueurs de vecteurs à sommer nécessitent un temps de calcul important, en particulier pour $n \geq 10^6$ (nous y reviendrons plus loin). Pour une première utilisation, on limite n aux valeurs 10^3 et 10^4.

```
VECT_SIZES = 1000 10000 # 100000 1000000
COND_SIZES = 32
DELTA_SIZES = 10 100 500 1000 1500 2000
```

Ces initialisations sont inclues dans le `Makefile`.

3.2.2 Makefile

Générons maintenant les exécutables nécessaires aux sections suivantes
Les exécutables des générateurs sont obtenus par

```
> make all_gen
```

Les exécutables des sommations et des tests sont obtenus par

```
> make all_sum
```

Ces commandes mettent à jour les répertoires `obj/` et `bin/`. Ce dernier doit maintenant contenir les exécutables suivants.

```
generator4 generator5dirac generator5unif
testprecsum testperfsum
```

Les trois premiers contruisent les vecteurs d'entrée à sommer dans `data`. Les exécutions s'effectueront avec `testprecsum` et `testperfsum`.

3.3 La génération des vecteurs à sommer et ses paramètres

Commande :
`> make gendata`

Fichier utilisé :
`generator4` dans `bin/`

Fichiers générés :
`n_cond.dat` dans `data/`

Il s'agit ici de générer des vecteurs d'entrée de longueurs variables $n = 10^3, 10^4, 10^5, 10^6$, et qui partagent des conditionnements du même ordre de grandeur, ici $cond = 10^{32}$.

Nous utilisons l'algorithme de génération de produits scalaires arbitrairement mal conditionnés proposé sous la forme d'un code Matlab dans [1]. Nous simplifions cet algorithme au cas de la sommation et proposons un codage en C décrit dans `gensum.h`.

```
// Conditionnning oriented : Ogita-Rump-Oishi's (2005)
double GenSum(double *x, double *C, unsigned int n, double c);
double GenSum2(double *x, double *C, unsigned int n, double exp_c);
```

Seul le dernier paramètre diffère selon ces deux versions : `c` est la valeur de *cond* souhaitée tandis que `exp_c` est son exposant (en base 10) ; *i.e.* `c` = $10^{\texttt{exp\_c}}$. Le générateur retourne la valeur effective `C` du conditionnement du vecteur généré qui peut être supérieure à la valeur souhaitée. Ensuite la commande
`> make gen_data`
génère un vecteur d'entrée pour chacune des valeurs de n et (de l'exposant) de *cond* indiqués dans le `make` [3]

Conseil. Regarder le code de `GenSum` pour comprendre le principe de génération.

Remarques.

- Ce générateur fait appel à une sommation très précise : AccSum dans [1]. D'autre part, générer de grands vecteurs peut prendre du temps sur une machine "ordinaire" : plusieurs heures pour $n = 10^6$ par exemple. Dans la version proposée, nous avons remplacé AccSum par OnLineExact.
- `GenSum` est suffisant dans cette section qui ne dépend pas fondamentalement de l'étendue des exposants — les performances de HybridSum et OnLineExact sont annoncées indépendantes de δ dans [2].
- Cependant l'étendue des exposants des vecteurs générés par `GenSum` ne permet pas de couvrir les grandes valeurs de δ du tableau 3.1. Nous avons modifié `GenSum` en `GenSumDelta` de façon à obtenir de telles valeurs pour des longueurs n multiples de 4.

```
// a la GenSum with a larger exponent range
double GenSumDelta(double *x, double *C, unsigned int n,
                   double exp_c, int delta);
```

3.4 Le calcul des sommes précises et fidèles

Commande :
`> make runsum`

Fichiers utilisés :
`testprecsum` dans `bin/`
`n_cond.dat` dans `data/`

Cette commande exécute les différents algorithmes de sommations sur les vecteurs d'entrée précédemment générés.

3.5 Analyse

On peut maintenant reprendre la démarche de la partie 1 pour l'analyse de la précision et de la reproductiblité (après un shuffle du vecteur d'entrée) de ces algorithmes/

3. Rappelons que pour les premières utilisation, n a été limité aux valeurs 1000 et 10000.

Références

[1] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput.*, 26(6) :1955–1988, 2005.

[2] Y.-K. Zhu and W. B. Hayes. Algorithm 908 : Online exact summation of floating-point streams. *ACM Trans. Math. Software*, 37(3) :37 :1–37 :13, Sept. 2010.