VERROU: Arrondis stochastiques déterministes

20-21/10/22
Réunion mi-ANR InterFLOP

Bruno Lathuilière (EDF R&D)

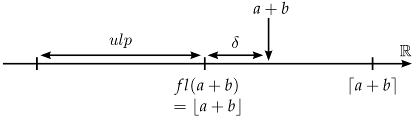
Travail en commun avec :
Nestor Demeure.

Plan

1. Les arrondis stochastiques et leurs limitations.
2. Les arrondis stochastiques déterministes
3. Recherche d'implémentations efficaces
4. Les arrondis stochastiques commutatifs déterministes
5. Perspectives

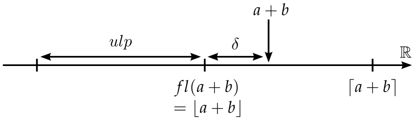
Modes d'arrondi `random` et `average` dans Verrou

- Transformation sans erreur :
  - $a \circ b = \sigma + \delta$,
  - $\sigma = fl(a \circ b)$



Modes d'arrondi `random` et `average` dans Verrou

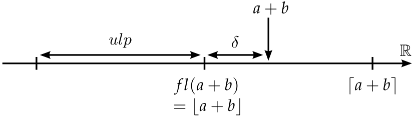
- Transformation sans erreur :
  - $a \circ b = \sigma + \delta$,
  - $\sigma = fl(a \circ b)$



- Si $\delta < 0$:
  $\lfloor a \circ b \rfloor = fl(a \circ b) - ulp$,
  $\lceil a \circ b \rceil = fl(a \circ b)$.
- Si $\delta = 0$:
  $\lfloor a \circ b \rfloor = fl(a \circ b)$
  $\lceil a \circ b \rceil = fl(a \circ b)$.
- Si $\delta > 0$:
  $\lfloor a \circ b \rfloor = fl(a \circ b)$,
  $\lceil a \circ b \rceil = fl(a \circ b) + ulp$.

Modes d'arrondi `random` et `average` dans Verrou

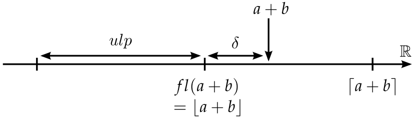
- Transformation sans erreur :
  - $a \circ b = \sigma + \delta$,
  - $\sigma = fl(a \circ b)$



- Si $\delta < 0$:
  $\lfloor a \circ b \rfloor = fl(a \circ b) - ulp$,
  $\lceil a \circ b \rceil = fl(a \circ b)$.
- Si $\delta = 0$:
  $\lfloor a \circ b \rfloor = fl(a \circ b)$
  $\lceil a \circ b \rceil = fl(a \circ b)$.
- **Si $\delta > 0$:**
  $\lfloor a \circ b \rfloor = fl(a \circ b)$,
  $\lceil a \circ b \rceil = fl(a \circ b) + ulp$.

Modes d'arrondi `random` et `average` dans Verrou

- Transformation sans erreur :
  - $a \circ b = \sigma + \delta$,
  - $\sigma = fl(a \circ b)$



- Si $\delta < 0$:
  $\lfloor a \circ b \rfloor = fl(a \circ b) - ulp$,

  $\lceil a \circ b \rceil = fl(a \circ b)$.

- Si $\delta = 0$:
  $\lfloor a \circ b \rfloor = fl(a \circ b)$

  $\lceil a \circ b \rceil = fl(a \circ b)$.

- **Si $\delta > 0$:**
  $\lfloor a \circ b \rfloor = fl(a \circ b)$,

  $\lceil a \circ b \rceil = fl(a \circ b) + ulp$.

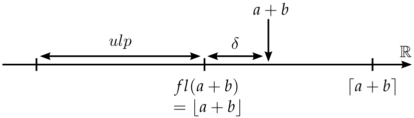
Mode `random` :

$$fl_{random}(a \circ b) = \left| \begin{array}{ll} \lfloor a \circ b \rfloor & avec\ p = 1/2 \\ \lceil a \circ b \rceil & avec\ p = 1/2 \end{array} \right.$$

Générateur pseudo aléatoire dans $\{0, 1\}$: (tinyMT ou xoshiro256plus)+ bit shift.

Modes d'arrondi `random` et `average` dans Verrou

- Transformation sans erreur :
  - $a \circ b = \sigma + \delta$,
  - $\sigma = fl(a \circ b)$



- Si $\delta < 0$:
  $\lfloor a \circ b \rfloor = fl(a \circ b) - ulp$,
  $\lceil a \circ b \rceil = fl(a \circ b)$.
- Si $\delta = 0$:
  $\lfloor a \circ b \rfloor = fl(a \circ b)$
  $\lceil a \circ b \rceil = fl(a \circ b)$.
- **Si $\delta > 0$:**
  $\lfloor a \circ b \rfloor = fl(a \circ b)$,
  $\lceil a \circ b \rceil = fl(a \circ b) + ulp$.

Mode `average` :

$$fl_{average}(a \circ b) = \left| \begin{array}{ll} \lfloor a \circ b \rfloor & avec\ p = \frac{1-\delta}{|ulp|} \\ \lceil a \circ b \rceil & avec\ p = \frac{\delta}{|ulp|} \end{array} \right.$$

Générateur pseudo aléatoire dans $\mathbb{F} \cap [0, 1]$: tinyMT ou xoroshiro128plus.
Equivalence avec MCA RR à la précision machine souvent appelé `SR_nearness`.

Problème avec les arrondis stochastiques

```
double a1=foo(42.);
double a2=foo(42.);
assert(a1==a2);
```

- Echec du assert

```
float x = foo (42);
if(x>0) return sqrt(foo(42));
else  return sqrt(-foo(42));
```

- NaN

```
class ProjectedCentralCircularSortOrder{
 ... constructor...
 bool operator()(const double* pt1, const double* pt2){
   const double ang1=atan2(pt1[_aIdx]-_a,pt1[_bIdx]-_b);
   const double ang2=atan2(pt2[_aIdx]-_a,pt2[_bIdx]-_b);
   return ang1 > ang2;}
}
```

```
ProjectedCentralCircularSortOrder order(...);
sort((polygon.begin()), polygon.end(), order);
```

- Erreur de segmentation

Remarque : problème similaire pour MCA mentionné par Stott Parker (paragraphe 6.8.2 "Handling multiple references properly" dans `http://fmdb.cs.ucla.edu/Treports/970002.pdf`)

Contournement et limitations

1. Ne pas pertuber ces fonctions :
   - Les erreurs commises dans ces fonctions sont ignorées.
2. Réécrire le code en stockant les calculs multiples.
   - Il n'est pas toujours facile/possible de modifier le code
   - Le nouveau code peut être moins performant.
3. Utiliser des clients request `VERROU_[START|STOP]_DETERMINISTIC`
   - Possible à l'échelle d'une fonction.
   - Peu utilisé dans la pratique.

Dans tous les cas on doit connaitre les fonctions concernées : cela nécessite l'usage de delta-debug puis de faire le tri entre les faux positifs et les vraies erreurs.

random_det et average_det

Idée : assurer le déterminisme interne à une exécution Verrou au niveau des opérations flottantes.
Moyen : remplacer le générateur pseudo aléatoire par une fonction de hashage qui prend en paramètre :

- `verrou_seed` : une graine de 64bit.
- `arg1, [arg2, [arg3]]` : les opérandes de l'opération.
- `Op` : le type de l'opération (enum désignant +,-,*,/, fma, cos, sin ...).

Pour `random_det` (respectivement `average_det`) l'espace d'arrivée de la fonction de hashage est $\{0, 1\}$ (respectivement $\mathbb{F} \cap [0, 1]$)
Souhait : conserver les mêmes propriétées que `random` (respectivement `average`), dans les cas où les opérandes ne répétent pas .

Avantage collatéral : le debug avec une seed fixée est facilité, avec la possibilité d'ajouter des opérations ne perturbant pas les variables d'intérêt (Exemple affichage de debug).

Implémentation naïve : mersenne twister

```
uint64_t mersenne_twister(uint64_t arg1,uint64_t arg2,
                          uint32_t Op){
  const uint64_t keys[4]={verrou_seed,arg1, arg2, Op};
  tinymt64_t gen;
  tinymt64_init_by_array(&gen, keys, 4);
  return tinymt64_generate_uint64(&gen);
}
```

hash function for `random_det` :

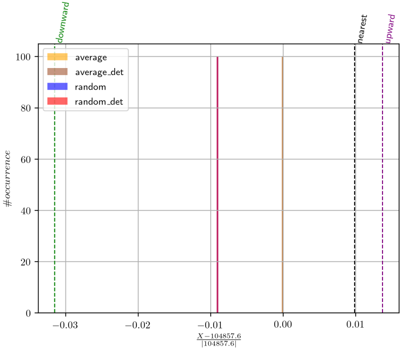
```
return mersenne_twister(arg1,arg2, Op) >> 63;
```

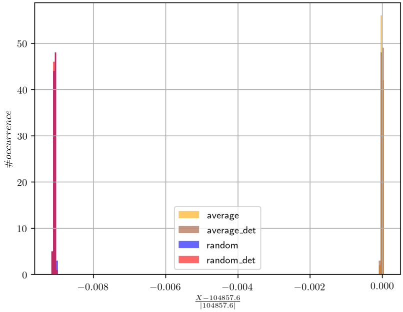
hash function for `average_det` :

```
const uint32_t v=mersenne_twister(arg1,arg2,Op)>>32;
constexpr double  invMax= (1./4294967296.);
return ((double)v * invMax );
```

Evaluation Seq

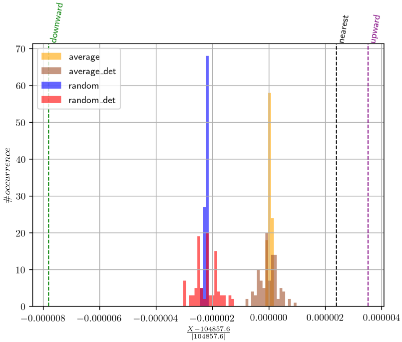
- Seq : sommation séquentielle de 2^{20} termes valant 0.1.
  - Comme l'accumulateur est différent à chaque étape: on espère que random et random_det soient similaires.





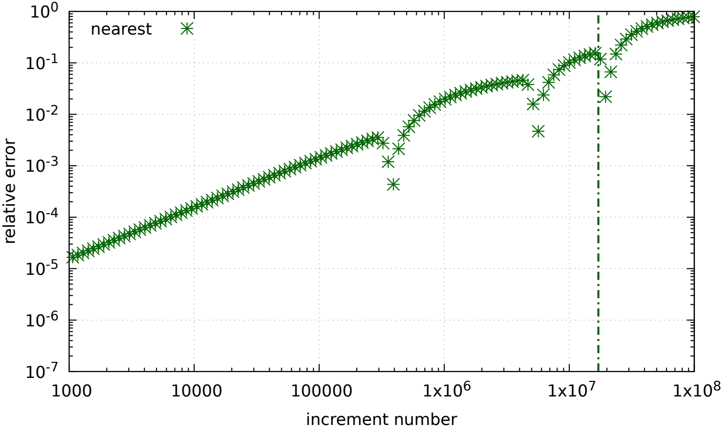
Evaluation `Rec`

- `Rec` : sommation récurssive de 2^{20} termes valant 0.1.
  - Récursion de base 4 : chaque tache est divisée en 4 sous-tâche.
  - Une tache de taille inférieure à 1024 éléments est calculée séquentiellement.
  - En base 2, sans seuil séquentiel, il n'y aurait pas d'erreur.



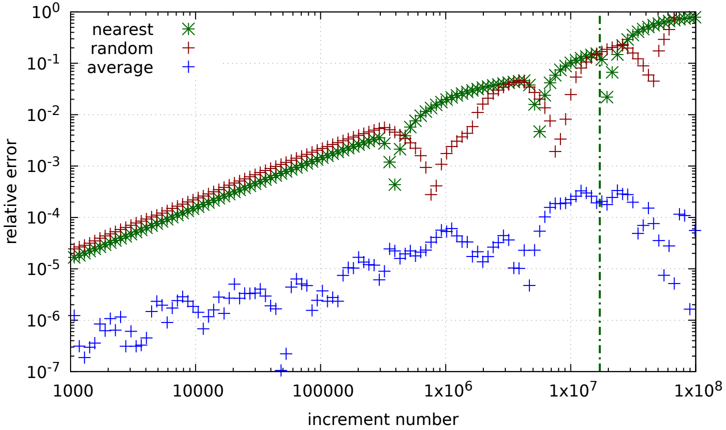
- **élargissement du support de la distribution observée.**

Effet sur la stagnation



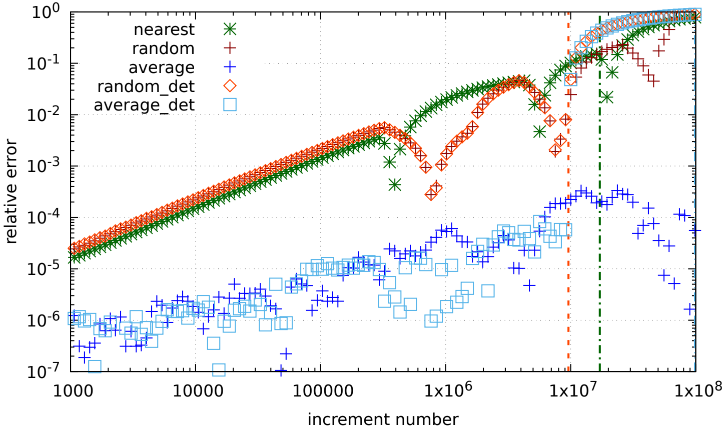
Error of accumulator (initialized to 100000) after *i* addition of 0.1. The vertical bars represent the begining of stagnation.

Effet sur la stagnation



Error of accumulator (initialized to 100000) after *i* addition of 0.1. The vertical bars represent the begining of stagnation.

Effet sur la stagnation



Error of accumulator (initialized to 100000) after *i* addition of 0.1. The vertical bars represent the begining of stagnation.

Recherche d'implémentations efficaces

	Testé	Gardé	Références
dietzfelbinger	Oui	Oui	[1 p.6]
multiply_shift	Oui	Oui	[1 Pair-Multiply-Shift p.15]
tabulation	Oui	Non	[2 p.9]
twisted_tabulation	Oui	Non	[2 p.9]
double_tabulation	Oui	Oui(par défaut)	[2]
MurmurHash3	Non	Non	Coûteux d'après [3]
tinyMT	Oui	Oui	

[1] M.Thorup. High Speed Hashing for Integers and Strings.
[2] M.Thorup. Fast and Powerful Hashing using Tabulation.

[3] S Dahlgaard, M.B.T Knudsen et M. Thorup. Practical Hash Function for Similarity Estimation and Dimensionality.

Verrou hash : dietzfelbinger implementation

```
uint64_t dietzfelbingerHash(uint64_t arg1,uint64_t arg2,
                            uint32_t Op){
  const uint64_t argsHash =  arg1 ^ arg2;
  const uint64_t seed = verrou_seed ^ (Op<<2);
  const uint64_t oddSeed = seed | 1;
return (oddSeed * argsHash);}
```

hash function for random_det:

```
return dietzfelbingerHash(arg1,arg2,Op) >> 63;
```

hash function for average_det:

```
const uint32_t res32(dietzfelbingerHash(arg1,arg2,Op)>>32);
constexpr double invMax=(1/ 4294967296.);
return ((double)res32 * invMax );
```

Verrou hash : multiplyShift

```
uint64_t multiplyShift(uint64_t arg1,uint64_t arg2,
                       uint32_t Op){
  uint32_t a1_1=arg1; uint32_t a1_2=arg1>>32;
  uint32_t a2_1=arg2; uint32_t a2_2=arg2>>32;

  return (a1_1+seedTab[0]) * (a1_2+seedTab[1])
       + (a2_1+seedTab[2]) * (a2_2+seedTab[3])
       + (Op*seedTab[6]) + seedTab[7];}
```

hash function for random_det :

```
return multiplyShift(arg1,arg2, Op) >> 63;
```

hash function for average_det :

```
const uint32_t v=multiplyShift(arg1,arg2,Op)>>32;
constexpr double  invMax= (1./4294967296.);
return ((double)v * invMax );
```

remarque : uint64_t seedTab[7] est généré par tinyMt initialisé avec verrou_seed.

Verrou hash : double_tabulation

```
void hash_aux(uint32_t& h,uint32_t index,uint64_t value){
  uint64_t x(value);
  for(int i=0 ; i <8 ; i++){
    uint8_t c=x;
    h^= hashTable[index][i][c];
    x = x >> 8;
  }}
uint32_t hash_op(uint16_t Op){
    uint32_t h=0;
    uint8_t c=Op; h^= hashTableOp[0][c];
    c = Op >> 8;  h^= hashTableOp[1][c];
    return h;}
uint32_t double_tabulation(uint64_t arg1,uint64_t arg2,
                           uint32_t Op){
   uint32 hash1=hash_op(res,Op);
   hash_aux(hash1, 0, arg1); hash_aux(hash1, 1, arg2);
   uint32 res=hash_aux(res, 3, hash1);
   return res;}
```

uint32_t hashTable[4][8][256] et uint32_t hashTableOp[2][256] sont initialisés par tinyMt avec verrou_seed.

Résultats : estimateurs sur 100 échantillons

	Seq		Rec	
	float	double	float	double
error(nearest)	6.66	35.92	18.67	47.92
all	4.61	34.05	16.58	45.99
random	5.73	36.05	17.68	47.92
random_det(dietzfelbinger)	5.10	36.06	17.30	47.09
random_det(multiply_shift)	4.85	34.41	16.90	46.43
random_det(double_tabulation)	5.73	36.05	17.36	47.59
random_det(mersenne_twister)	5.73	36.06	17.39	47.32
average	6.67	35.91	18.63	47.92
average_det(dietzfelbinger)	6.10	35.92	17.95	46.90
average_det(multiply_shift)	4.99	34.24	17.02	46.32
average_det(double_tabulation)	6.67	35.91	18.19	47.32
average_det(mersenne_twister)	6.67	35.91	18.25	47.32

$s_{random} = -log2\left(\frac{max_{i \in random}(|x_i - x_{nearest}|)}{|x_{nearest}|}\right)$ $\quad error(nearest) = -log2\left(\frac{|x_{ref} - x_{nearest}|}{|x_{ref}|}\right)$

$s_{all} = max(s_{random}, s_{average}, s_{downward}, s_{upward})$

Résultats : performance

Programme: stencil en float/double compilé en O0/O3

type	double		float	
compilation option	O0	O3	O0	O3
nearest	x15.2	x35.3	x18.1	x49.0
random	x22.2	x61.0	x27.6	x98.4
random_det(dietzfelbinger)	x22.5	x61.4	x27.5	x96.6
random_det(multiply_shift)	x22.9	x62.8	x27.5	x96.4
random_det(double_tabulation)	x27.0	x79.0	x31.6	x118.8
random_det(mersenne_twister)	x42.0	x134.3	x54.9	x226.9
average	x25.0	x70.7	x31.1	x111.8
average_det(dietzfelbinger)	x24.2	x65.1	x29.8	x100.2
average_det(multiply_shift)	x24.6	x66.4	x30.1	x101.1
average_det(double_tabulation)	x29.6	x85.6	x35.3	x127.2
average_det(mersenne_twister)	x44.7	x140.2	x59.2	x235.1

Attention : à refaire passer avec la dernière version de verrou.

Arrondis stochastiques commutatifs déterministes

Problème :

```
assert(dot(x,y)==dot(y,x))
```

Solution : Introduction des modes `[random,average]_comdet` qui garantissent que $x\ op\ y$ soient arrondis comme $y\ op\ x$.

Implémentation :

- Pour dietzfelbinger `random_det` a déjà cette propriété.
- Pour les autres on remplace hash(arg1,arg2,Op) par hash(min(arg1,arg2),max(arg1,arg2),Op).

Performance :
Pour l'instant très faible impact (dans le bruit de mesure).

Conclusions et perspectives

Conclusion

- [random,average]_det suppriment des faux-positifs sans besoin de modifier le code.
- [random,average]_det simplifient le deboguage avec une graine fixée.
- [random,average]_det ont un surcoût acceptable vis à vis de [random,average].

Perspectives

- Optimisation du backend verrou pour random/average (xo[ro]shiro, ...).
- Optimisation du frontend valgrind/verrou.
- Tester xo[ro]shiro pour [random,average]_det.
- Généralisation pour MCA (PB, RR, FULL).
- Test du remplacement du mode average[_det] en float par SR_nearness.
- Besoin de REX sur les résultats du delta-debug en mode [random,average]_[det,comdet].