VERROU: Nouveaux modes d'arrondi stochastique pour la vérification numérique

09/06/23
Réunion InterFLOP

Bruno Lathuilière (EDF R&D)

Travail en commun avec :
Nestor Demeure.

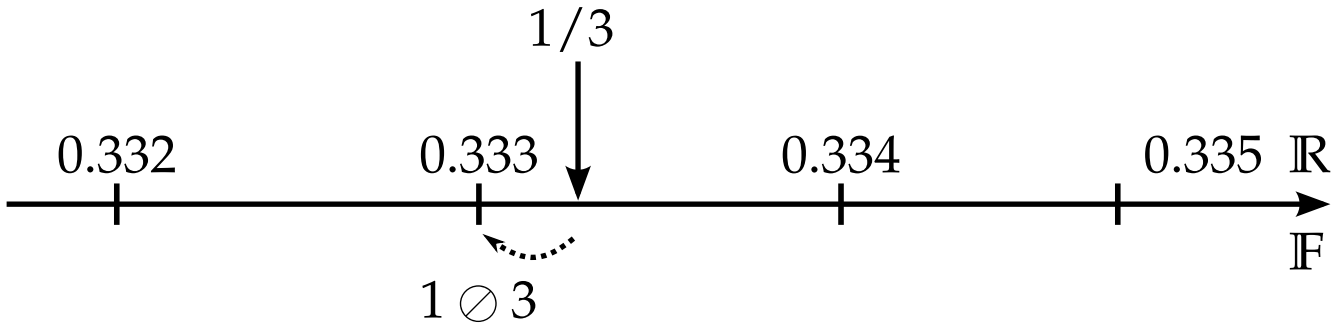
Plan

1. Contexte
2. Les arrondis stochastiques et leurs limitations.
3. Les arrondis stochastiques déterministes
4. Recherche d'implémentations efficaces
5. Les arrondis stochastiques commutatifs déterministes
6. Arrondis stochastiques monotones
7. Perspectives
8. Bonus : prandom

Arithmétique stochastique

Modéliser l'imprécision par un aléa sur le mode d'arrondi

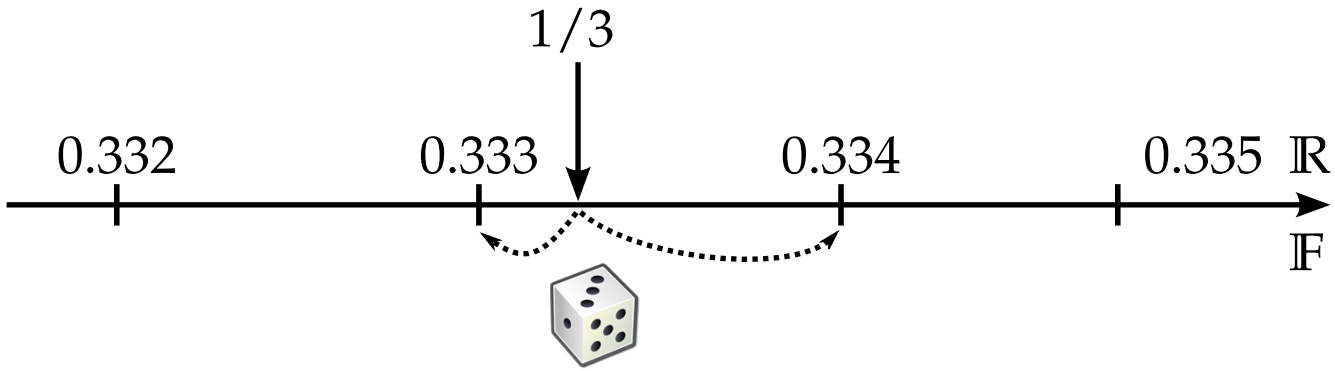
Arrondi au plus proche (défaut IEEE-754)



Arithmétique stochastique

Modéliser l'imprécision par un aléa sur le mode d'arrondi

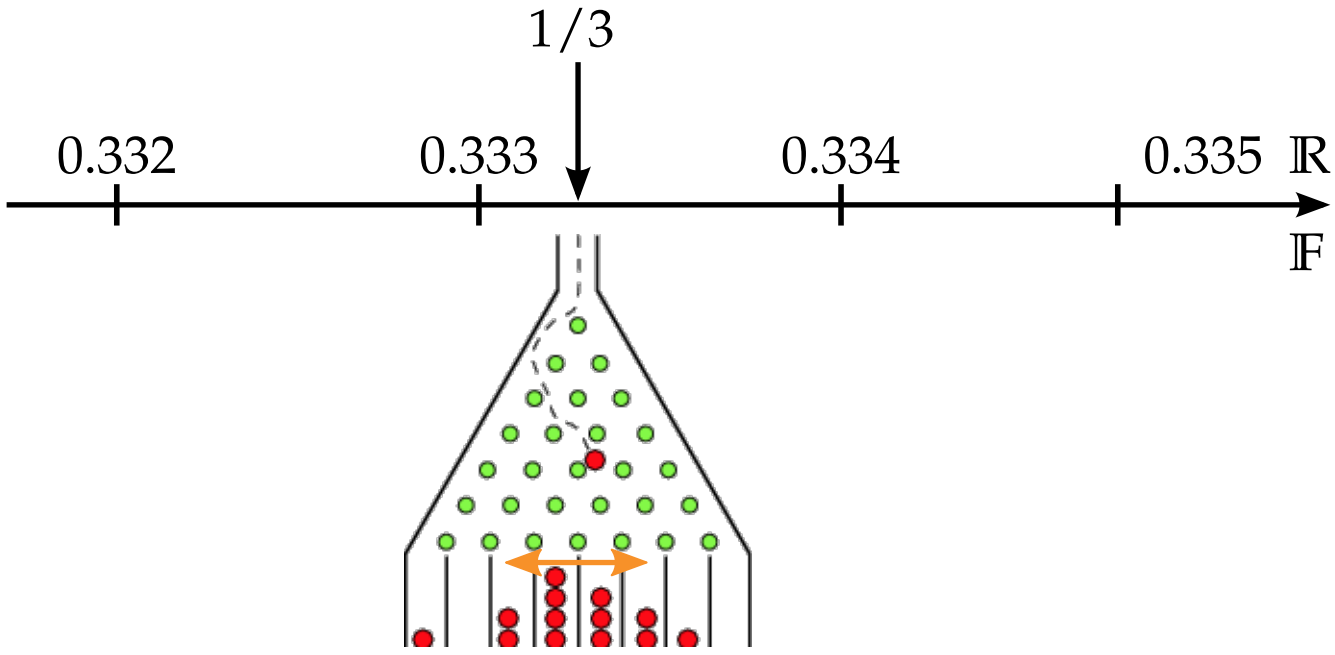
Arrondi aléatoire



Arithmétique stochastique

Modéliser l'imprécision par un aléa sur le mode d'arrondi

Arrondi aléatoire



Instruction	Eval. 1	Eval. 2	Eval. 3	
$a = 1/3$	$0.333_{\downarrow}$	$0.334^{\uparrow}$	$0.334^{\uparrow}$	
$b = a \times 3$	0.999	$1.00_{\downarrow}$	$1.01^{\uparrow}$	**1.00**

Faux positifs avec les arrondis stochastiques

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

- Echec du assert

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

- NaN

```
1 class ProjectedCentralCircularSortOrder{
2  ... constructor...
3  bool operator()(const double* pt1, const double* pt2){
4    const double ang1=atan2(pt1[_aIdx]-_a,pt1[_bIdx]-_b);
5    const double ang2=atan2(pt2[_aIdx]-_a,pt2[_bIdx]-_b);
6    return ang1 > ang2;}
7 }
```

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

- Erreur de segmentation

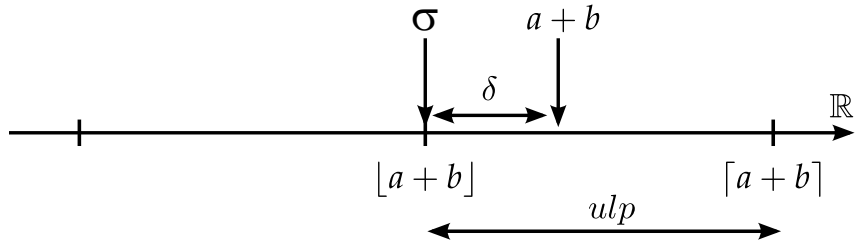
Contournements actuels

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.

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.

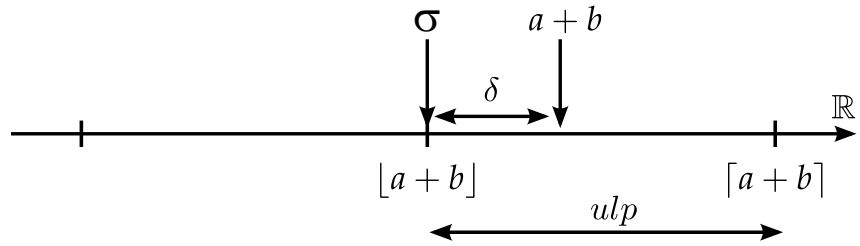
Verrou : implémentation des arrondis stochastiques

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



Verrou : implémentation des arrondis stochastiques

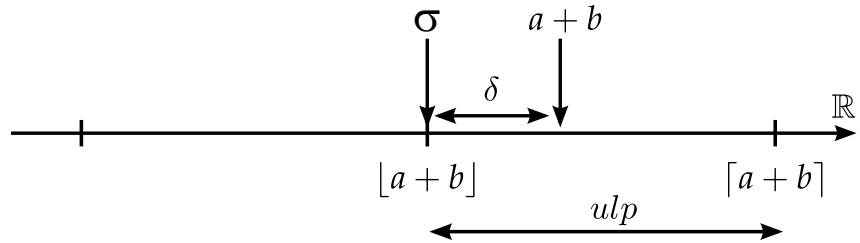
- Transformation sans erreur :
  - $a \circ b = \sigma + \delta$,
  - $\sigma = fl(a \circ b)$



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

Verrou : implémentation des arrondis stochastiques

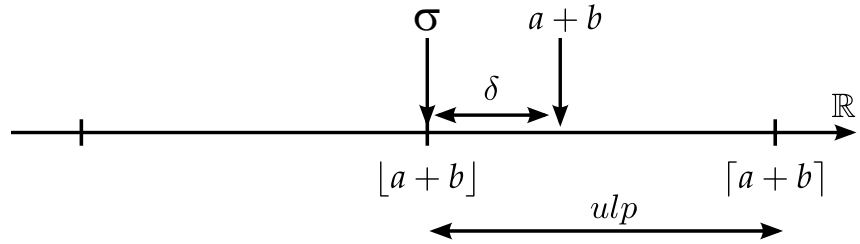
- Transformation sans erreur :
  - $a \circ b = \sigma + \delta$,
  - $\sigma = fl(a \circ b)$



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

Verrou : implémentation des arrondis stochastiques

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



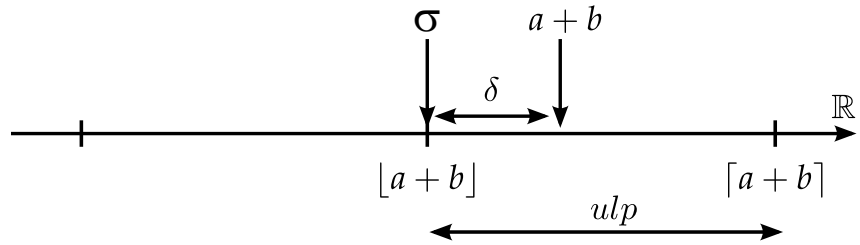
- Si $\delta < 0$:
  $\lfloor a \circ b \rfloor = fl(a \circ b) - ulp$,
  $\lceil a \circ b \rceil = fl(a \circ b)$.
- Si $\delta = 0$:
  $\lfloor a \circ b \rfloor = fl(a \circ b)$
  $\lceil a \circ b \rceil = fl(a \circ b)$.
- **Si $\delta > 0$:**
  $\lfloor a \circ b \rfloor = fl(a \circ b)$,
  $\lceil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

Mode `random` : $$fl_{random}(a \circ b) = \begin{cases} \lfloor a \circ b \rfloor & avec\ p = 1/2 \\ \lceil a \circ b \rceil & avec\ p = 1/2 \end{cases}$$

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

Verrou : implémentation des arrondis stochastiques

- 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`.

`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 $\oplus$,$\ominus$,$\otimes$,$\oslash$, *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 se répétent pas .

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};
  const uint64_t keys[3]={verrou_seed^Op,arg1, arg2};
  tinymt64_t gen;
  tinymt64_init_by_array(&gen, keys, 3);
  return tinymt64_generate_uint64(&gen);
}
```

hash function for random_det :

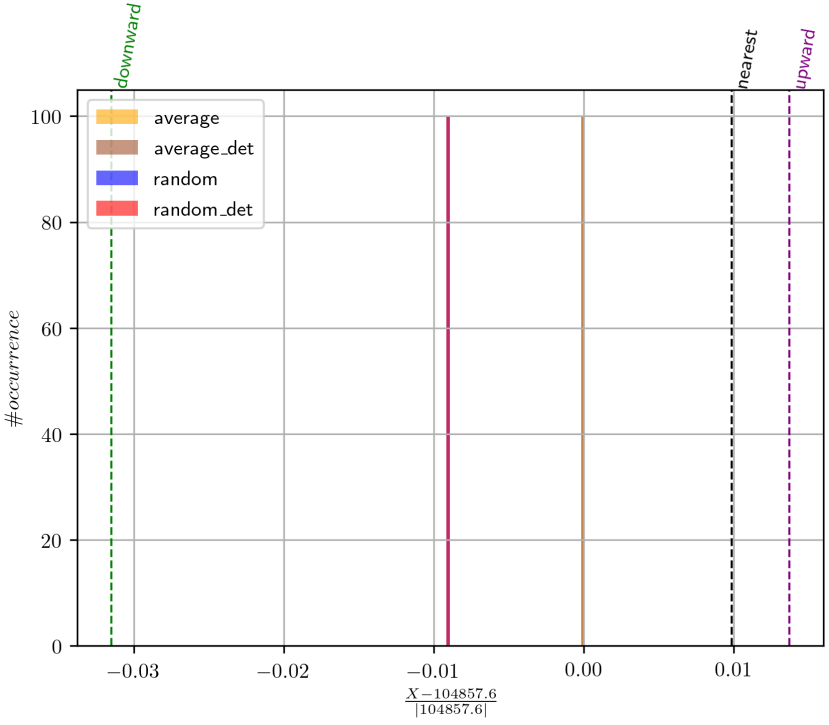
```
return mersenne_twister(arg1,arg2, Op) >> 63;
```

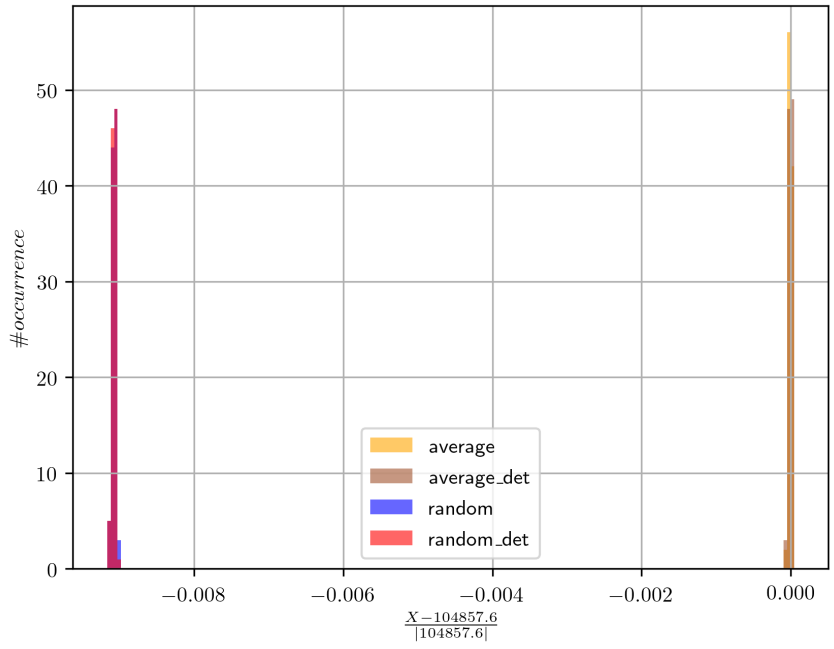
hash function for average_det :

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

Evaluation Seq

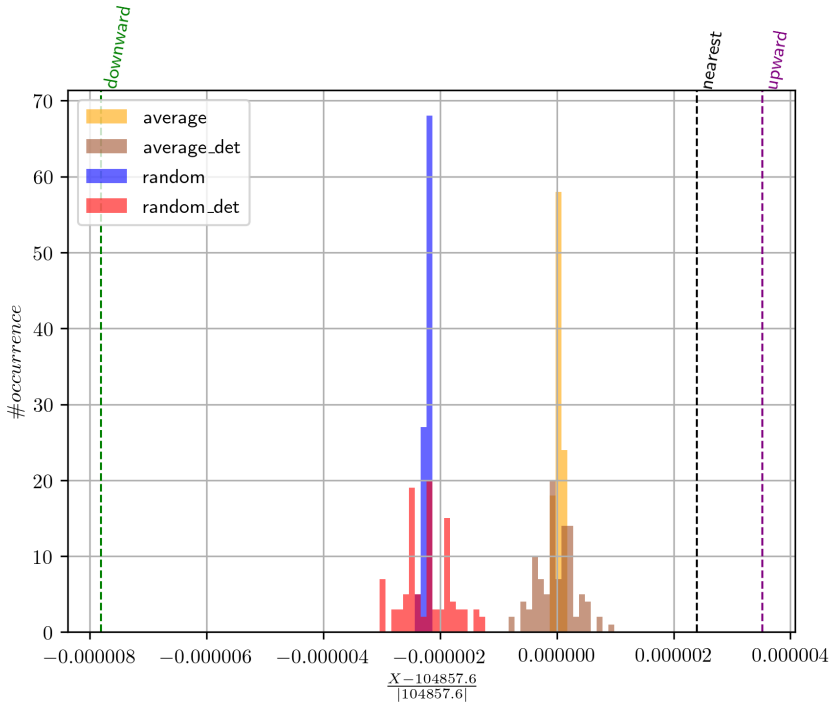
- Seq : sommation séquentiel de 2^{20} termes valant 0.1.
  - Comme l'accumulateur est différent à chaque étape: on s'attend à ce que `random` et `random_det` soient similaires.





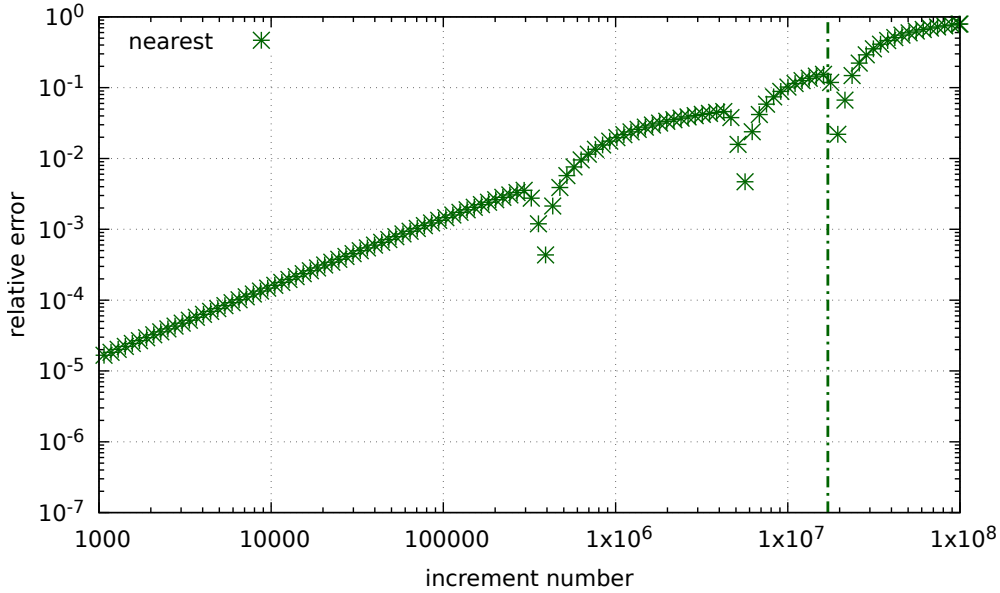
Evaluation `Rec`

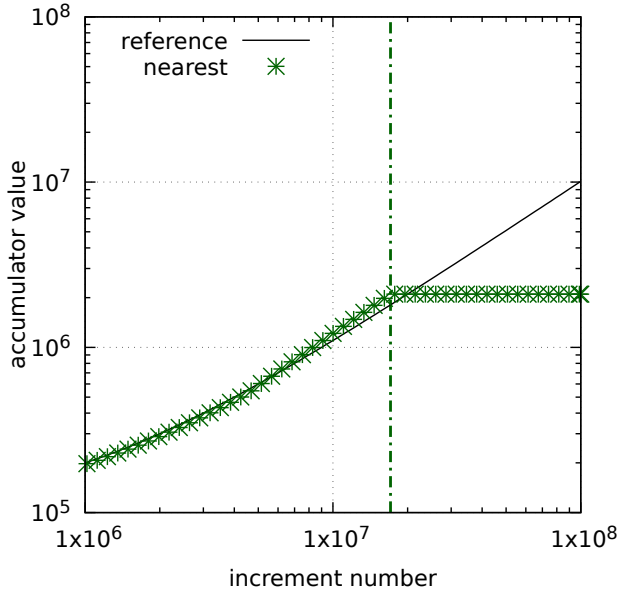
- `Rec` : sommation récurssive de 2^{20} termes valant 0.1.
  - Récursion de base 4 : chaque tache est divisée en 4 sous-tâches.
  - Une tache de taille inférieure à 1024 éléments est calculée séquentiellement.
  - En base 2, sans seuil séquentiel, il n'y aurait pas d'erreur.



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

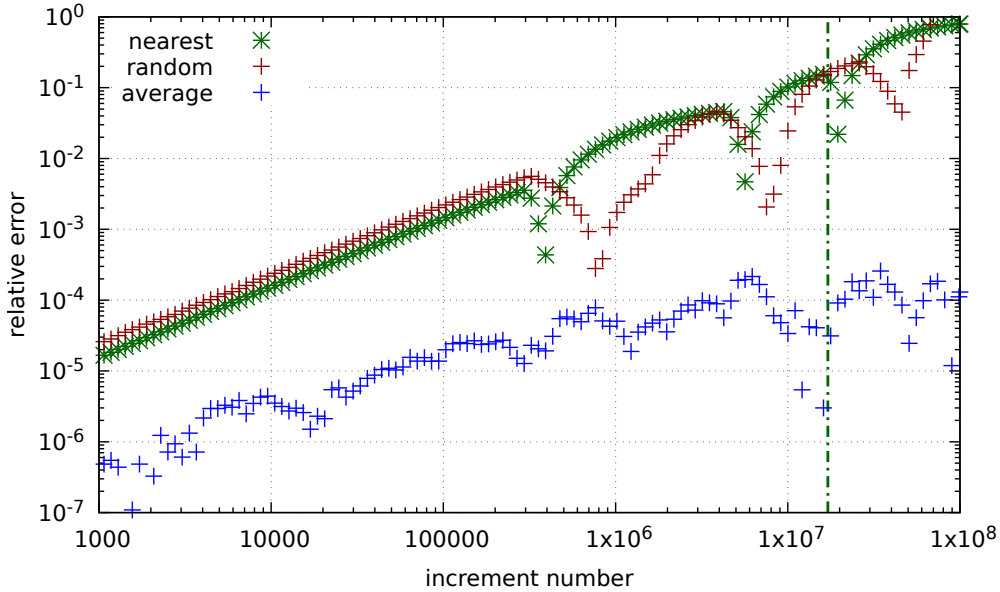
Effet sur la stagnation : incrément constant

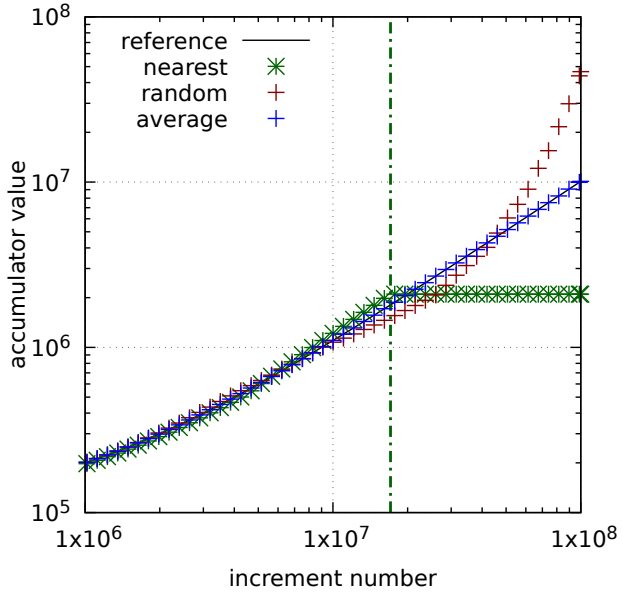




Error (and value) of accumulator (initialized to 100000) after i additions of 0.1.
The vertical bars represent the begining of stagnation.

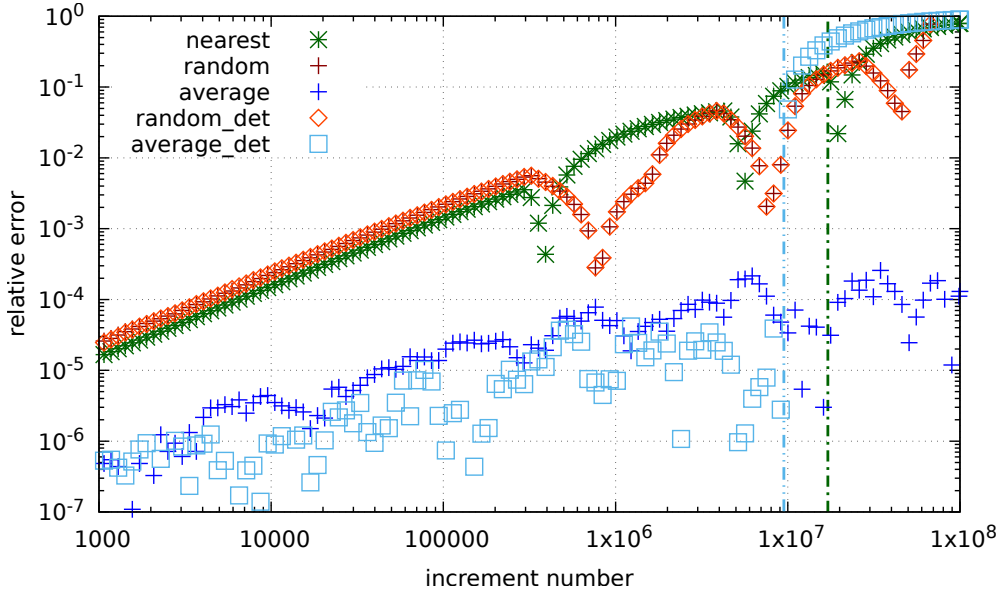
Effet sur la stagnation : incrément constant

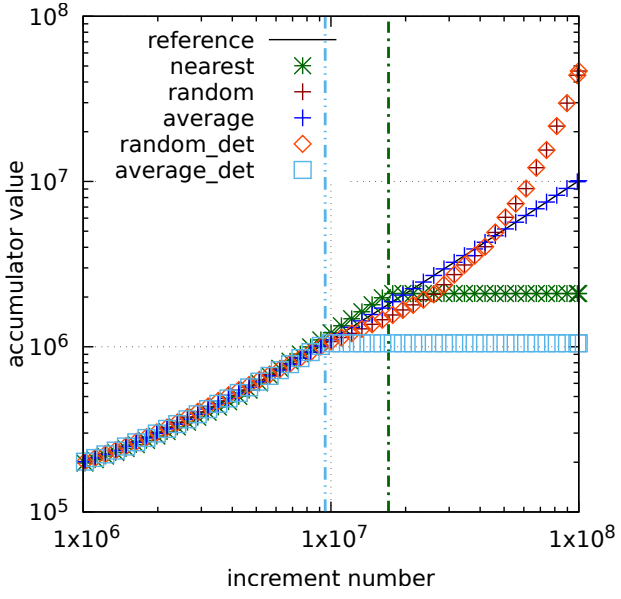




Error (and value) of accumulator (initialized to 100000) after i additions of 0.1.
The vertical bars represent the begining of stagnation.

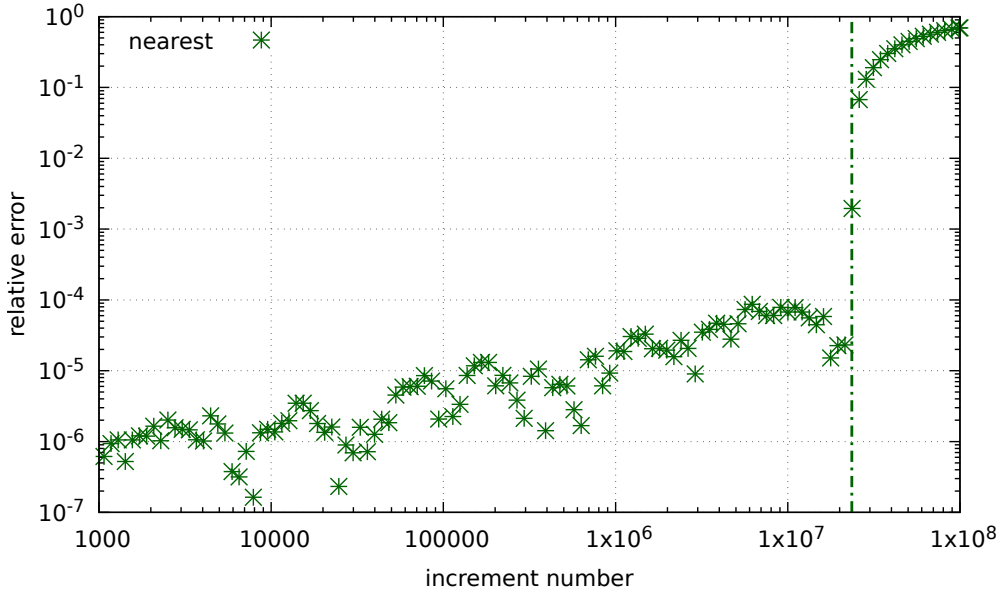
Effet sur la stagnation : incrément constant

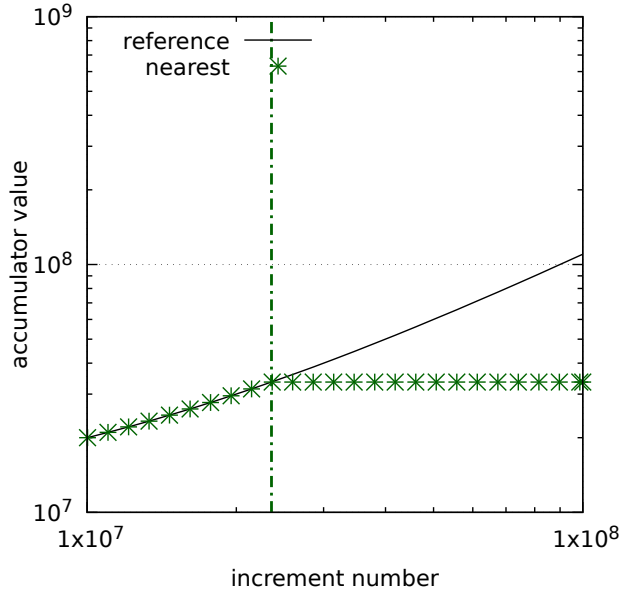




Error (and value) of accumulator (initialized to 100000) after i additions of 0.1.
The vertical bars represent the begining of stagnation.

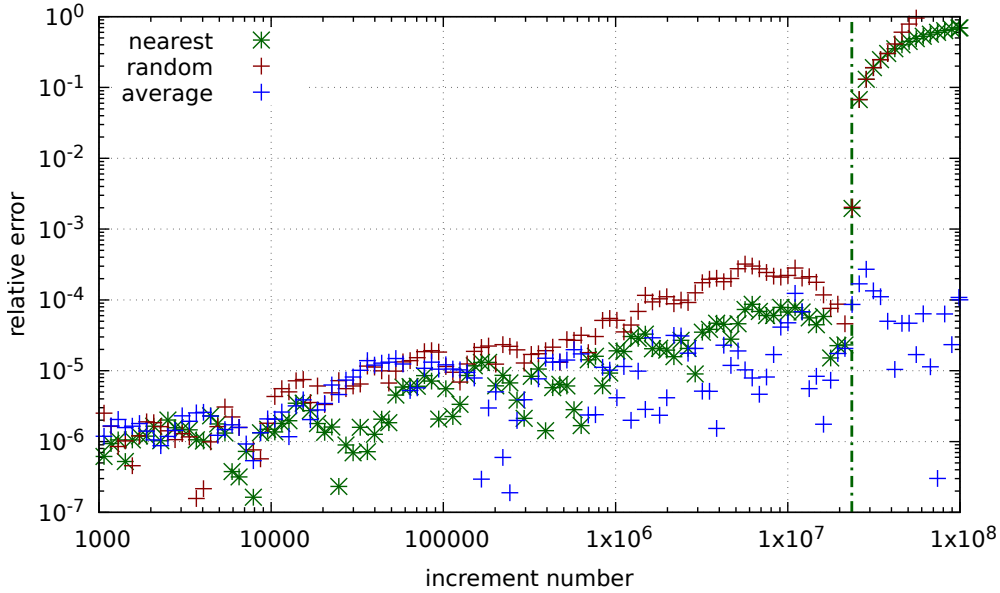
Effet sur la stagnation : incrément random

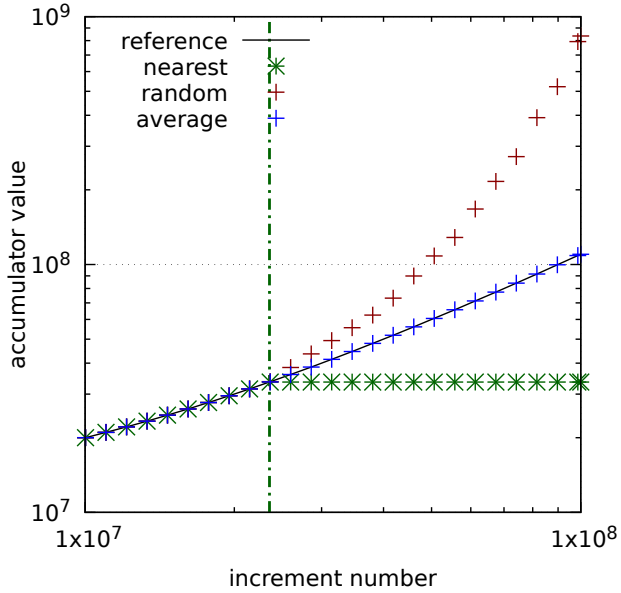




Error (and value) of accumulator (initialized to 10000000) after i additions of random value (uniform distribution between 0 and 2)

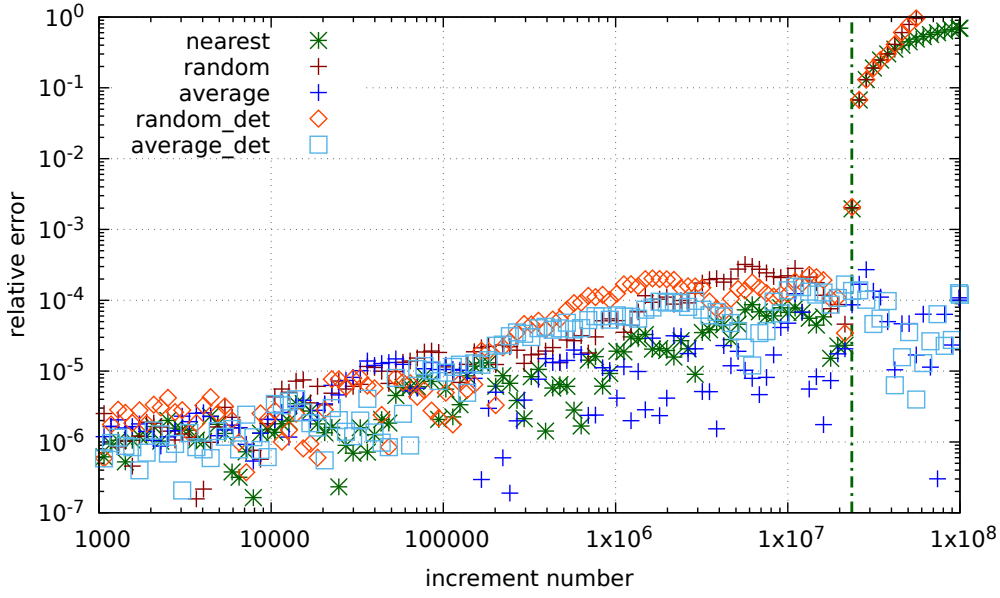
Effet sur la stagnation : incrément random

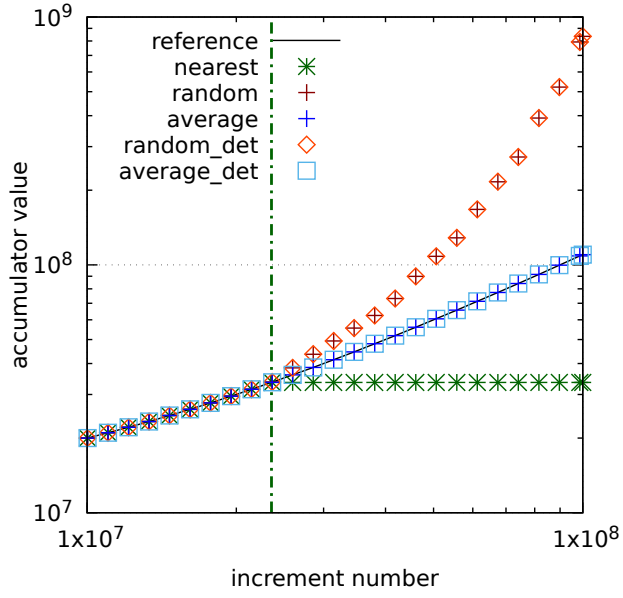




Error (and value) of accumulator (initialized to 10000000) after i additions of random value (uniform distribution between 0 and 2)

Effet sur la stagnation : incrément random





Error (and value) of accumulator (initialized to 10000000) after i additions of random value (uniform distribution between 0 and 2)

Recherche d'implémentations efficaces

	Espace de départ et d'arrivée	Références
dietzfelbinger	uint64 $\rightarrow$ uint32	[1 p.6]
multiply_shift	multiple de uint32 $\rightarrow$ uint32	[1 Pair-Multiply-Shift p.15]
double_tabulation	multiple de uint8 $\rightarrow$ uint32	[2]
xxhash(xxh3)	multiple de uint8 $\rightarrow$ uint64	[3]
tinyMT	muliple de uint64 $\rightarrow$ uint64	

[1] M.Thorup. High Speed Hashing for Integers and Strings.
[2] M.Thorup. Fast and Powerful Hashing using Tabulation.
[3] `https://github.com/Cyan4973/xxHash`

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

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(double_tabulation)	5.73	36.06	17.41	47.32
random_det(xxhash)	**5.73**	**36.05**	**17.41**	**47.32**
random_det(mersenne_twister)	5.73	36.05	**17.59**	**47.32**
average	**6.67**	**35.91**	18.63	47.92
average_det(double_tabulation)	6.67	35.91	18.19	47.32
average_det(xxhash)	**6.67**	**35.91**	**18.32**	**47.32**
average_det(mersenne_twister)	6.67	35.91	**18.32**	**47.59**

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

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

Résultats : performance

Programme: stencil en float/double compilé en O0/O3 (avec fma)

type	double		float	
compilation option	O0	O3	O0	O3
nearest	x11.9	x24.9	x11.7	x32.4
random	**x18.8**	**x52.2**	**x19.2**	**x77.6**
random_det(double_tabulation)	x22.8	x66.4	x21.6	x93.3
random_det(xxhash)	**x19.2**	**x51.7**	**x19.6**	**x80.5**
random_det(mersenne_twister)	x36.9	x115.6	x39.4	x204.6
average	**x21.7**	**x60.1**	**x22.2**	**x92.0**
average_det(double_tabulation)	x26.8	x80.5	x26.5	x116.7
average_det(xxhash)	**x23.4**	**x65.4**	**x24.5**	**x102.2**
average_det(mersenne_twister)	x41.0	x127.5	x44.7	x228.1

Arrondis stochastiques commutatifs déterministes

Problème :

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

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

Implémentation :

- Pour dietzfelbinger `random_det` a déjà cette propriété.
- Pour les autres on remplace :
  - $hash(arg1, arg2, op)$ par $hash(min(arg1, arg2), max(arg1, arg2), op)$ pour les opérateurs commutatifs ($\oplus$,$\otimes$).
  - $hash(arg1, arg2, arg3, FmaEnum)$ par $hash(min(arg1, arg2), max(arg1, arg2), arg3, FmaEnum)$.

Arrondis stochastiques commutatifs déterministes

Variante signée

Problème :

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

Solution : Introduction des modes `[random,average]_scomdet` qui garantissent la commutativité des opérateurs commutatifs et :
$\forall a, b, c \in \mathbb{F}^3$

- $(a \oplus b) = -(-a \oplus (-b))$
- $(a \oplus (-b)) = -(-a \oplus b)$
- $(a \ominus b) = (a \oplus (-b))$
- $(a \otimes b) = ((-a) \otimes (-b))$
- $(a \otimes b) = -(a \otimes (-b))$
- $(a \oslash b) = ((-a) \oslash (-b))$
- $(a \oslash b) = -(a \oslash (-b))$
- $(a \oslash b) = -(-a \oslash (b))$

- $fma(a, b, c) = fma(-a, -b, c)$
- $fma(-a, b, -c) = -fma(a, b, c)$
- $fma(a, -b, -c) = -fma(a, b, c)$
- $cast(a) = -cast(-a)$

Bilan des contraintes sur les hash

Soit $x, y, z, a, b \in \mathbb{F}^5$ *avec* $a > 0, b > 0$

OP x	OP y	$\oplus$	$\ominus$	$\otimes$	$\oslash$	fma z>0	fma z=0	fma z<0
a	b	$r_{\oplus}$	$r_{\ominus}$	$r_{\otimes}$	$r^1_{\oslash}$	r^1_{fma}	$r_{\otimes}$	$\overline{r^2_{fma}}$
b	a	$r_{\oplus}$	$\overline{r_{\ominus}}$	$r_{\otimes}$	$r^2_{\oslash}$	r^1_{fma}	$r_{\otimes}$	$\overline{r^2_{fma}}$
-a	-b	$\overline{r_{\oplus}}$	$\overline{r_{\ominus}}$	$r_{\otimes}$	$r^1_{\oslash}$	r^1_{fma}	$r_{\otimes}$	$\overline{r^2_{fma}}$
-b	-a	$\overline{r_{\oplus}}$	$r_{\ominus}$	$r_{\otimes}$	$r^2_{\oslash}$	r^1_{fma}	$r_{\otimes}$	$\overline{r^2_{fma}}$
a	-b	$r_{\ominus}$	$r_{\oplus}$	$\overline{r_{\otimes}}$	$\overline{r^1_{\oslash}}$	r^2_{fma}	$\overline{r_{\otimes}}$	$\overline{r^1_{fma}}$
-b	a	$r_{\ominus}$	$\overline{r_{\oplus}}$	$\overline{r_{\otimes}}$	$\overline{r^2_{\oslash}}$	r^2_{fma}	$\overline{r_{\otimes}}$	$\overline{r^1_{fma}}$
-a	b	$\overline{r_{\ominus}}$	$\overline{r_{\oplus}}$	$\overline{r_{\otimes}}$	$\overline{r^1_{\oslash}}$	r^2_{fma}	$\overline{r_{\otimes}}$	$\overline{r^1_{fma}}$
b	-a	$\overline{r_{\ominus}}$	$r_{\oplus}$	$\overline{r_{\otimes}}$	$\overline{r^2_{\oslash}}$	r^2_{fma}	$\overline{r_{\otimes}}$	$\overline{r^1_{fma}}$

$\overline{r} = 1 - r$: si r conduit à un arrondi `upward` alors $\overline{r}$ conduit à un arrondi `downward` (et reciproquement)

Les cas x=0 et y=0 sont non traités car ils conduisent à des opérations exactes.

$r_{op} = hash(min(|x|, |y|), max(|x|, |y|), op) \quad \forall op \in \{\oplus, \otimes, \ominus\}$

$r^1_{\oslash} = hash(|x|, |y|, \oslash)$ et $r^2_{\oslash} = hash(|x|, -|y|, \oslash)$

$r^1_{fma} = hash(min(|x|, |y|), max(|x|, |y|), |z|, fma)$ et

$r^2_{fma} = hash(min(|x|, |y|), max(|x|, |y|), -|z|, fma)$

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_scomdet(xxhash)	**5.73**	**36.05**	**17.49**	**47.32**
random_scomdet(mersenne_twister)	5.73	36.05	**17.45**	**47.32**
average	**6.67**	**35.91**	18.63	47.92
average_scomdet(xxhash)	**6.67**	**35.91**	**18.19**	**47.32**
average_scomdet(mersenne_twister)	6.67	35.91	**18.19**	**47.32**

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

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

Arrondis stochastiques monotones

Problème :

```
1 assert((x <= y) == (a+x <= a+y) )
```

Solution : Introduction du mode `sr_monotonic` :

$$fl_{monotonic}(a \circ b) = \begin{cases} \lfloor a \circ b \rfloor & si\ a \circ b < seuil \\ \lceil a \circ b \rceil & si\ a \circ b > seuil \\ \lfloor a \circ b \rfloor & si\ a \circ b = seuil \quad et \quad a \circ b < 0 \\ \lceil a \circ b \rceil & si\ a \circ b = seuil \quad et \quad a \circ b > 0 \end{cases}$$

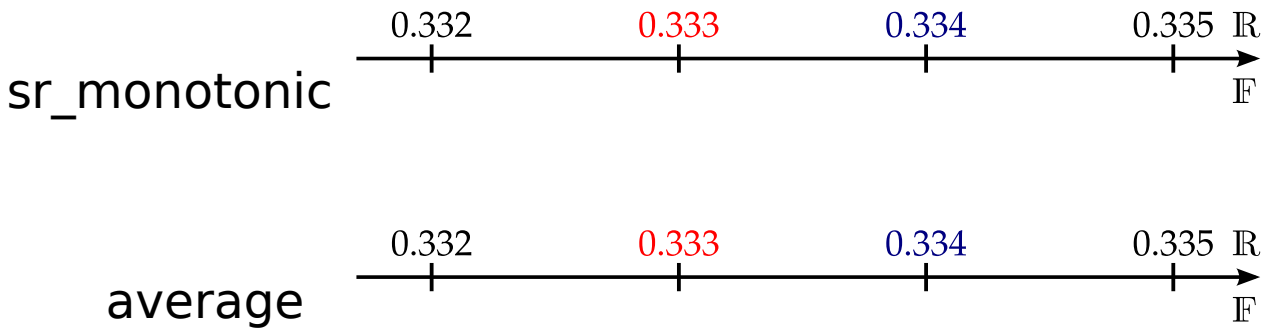
avec *seuil* échantillonné (une fois par exécution du programme) suivant une loi uniforme dans $[\lfloor a \circ b \rfloor, \lceil a \circ b \rceil]$ tel que

$$seuil([\lfloor -(a \circ b) \rfloor, \lceil -(a \circ b) \rceil]) = -seuil([\lfloor a \circ b \rfloor, \lceil a \circ b \rceil])$$

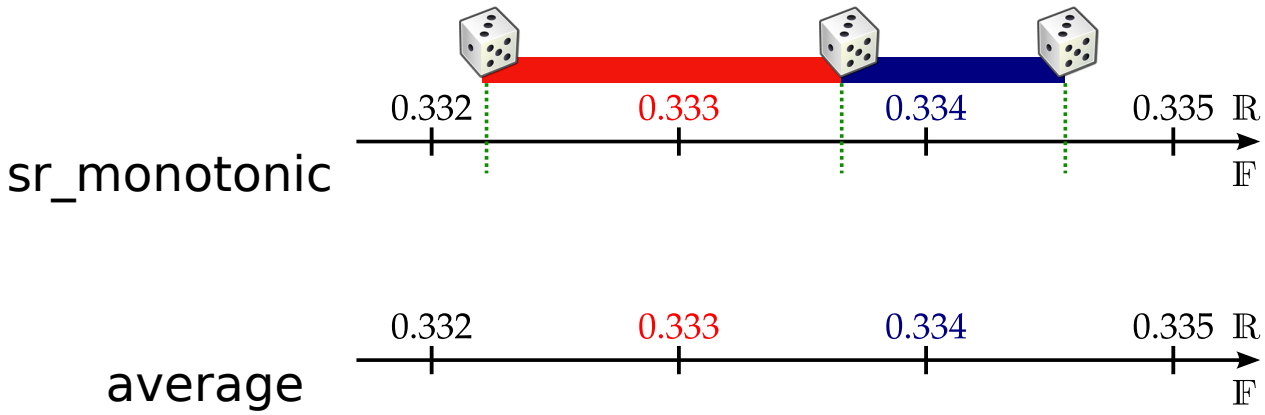
Remarque :

- la contrainte de parité du seuil permet de respecter les contraintes de `[average|random]_scomdet`.
- dans la pratique *seuil* est obtenu par le produit de *ulp* par la fonction de hashage dans $[0., 1]$ prennant en paramètre l'arrondi vers 0 de $|a \circ b|$.

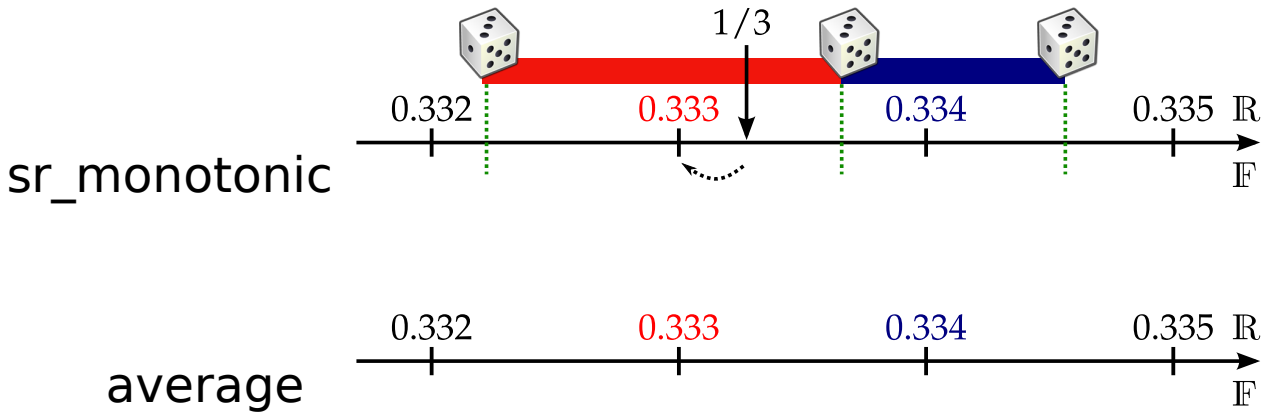
Comparaison à average



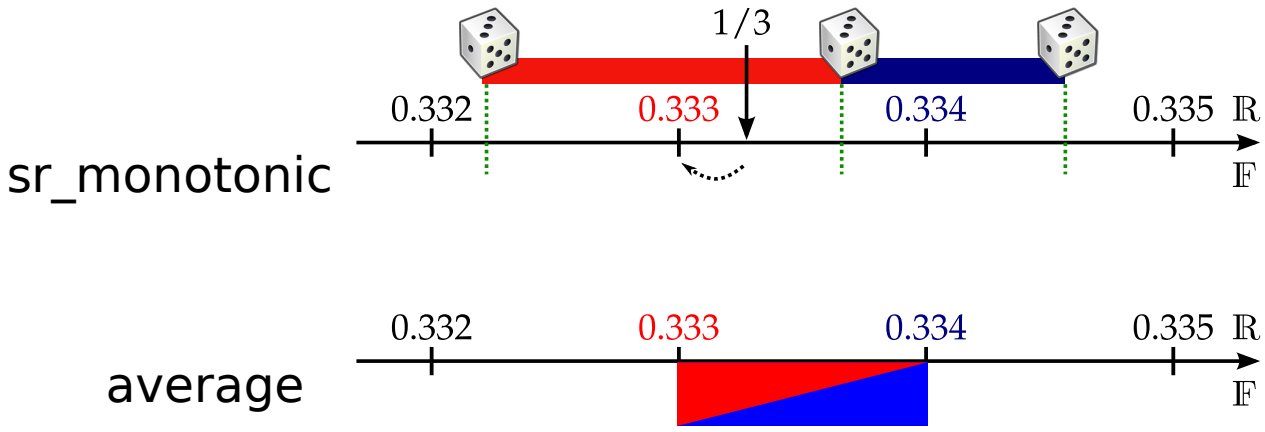
Comparaison à average



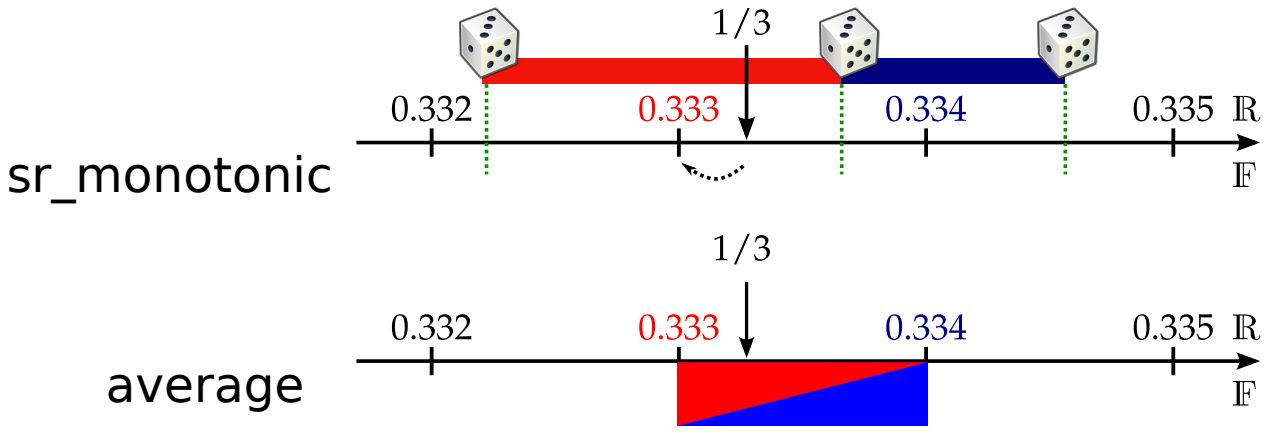
Comparaison à average



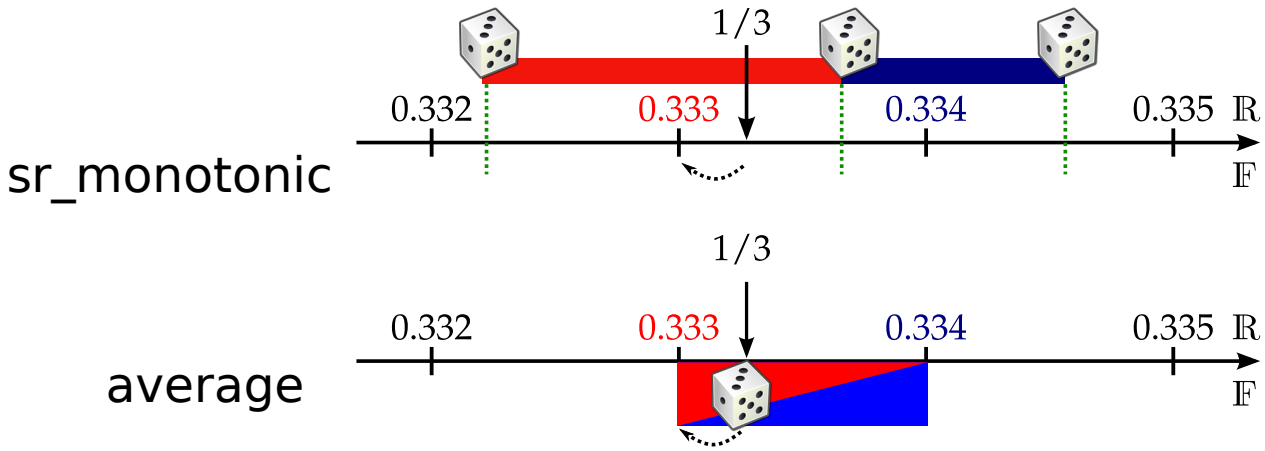
Comparaison à average



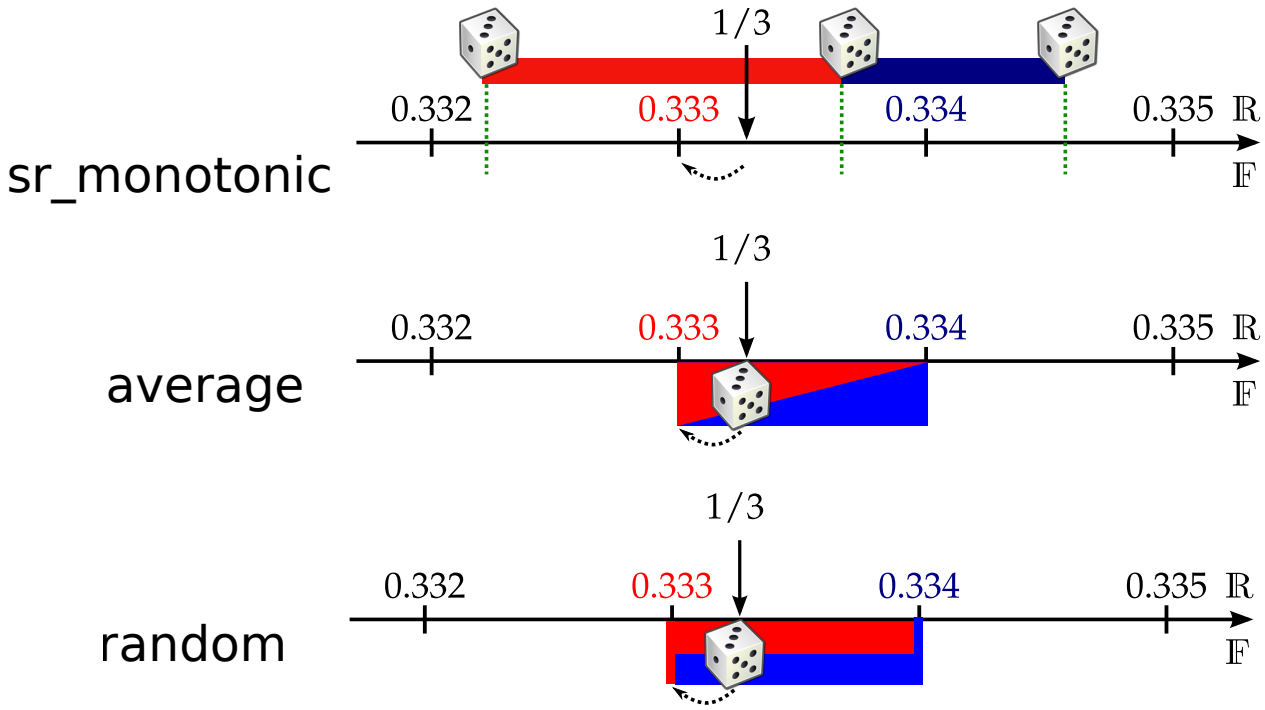
Comparaison à average



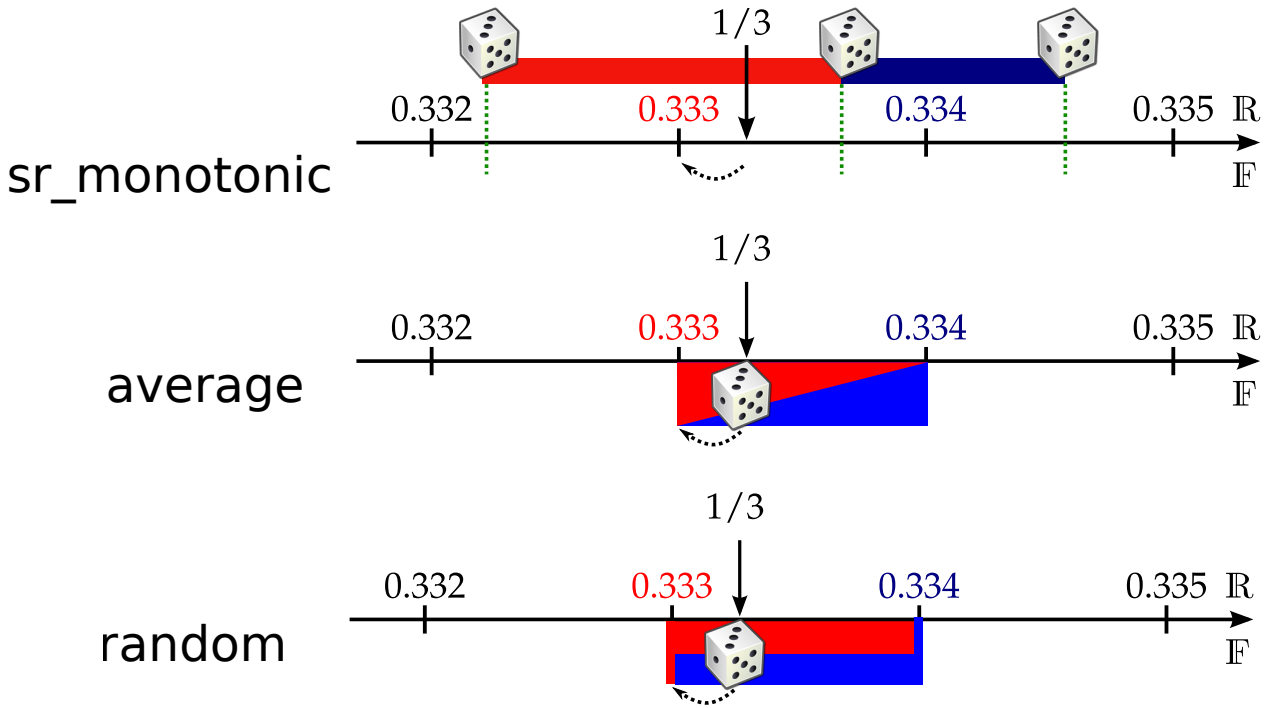
Comparaison à average



Comparaison à average



Comparaison à average



Remarque : Vis-à-vis de `average` :

- on perd l'indépendance des tirages aléatoires,
- mais si les résultats de toutes les opérations sont dans des intervalles différents, on a équivalence.

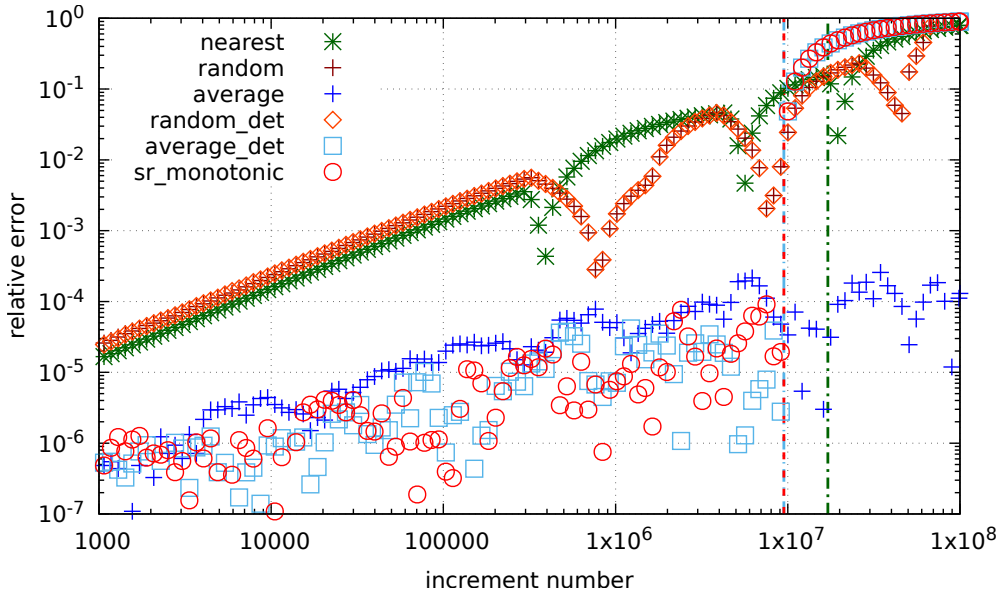
Résultats : estimateurs sur 100 échantillons

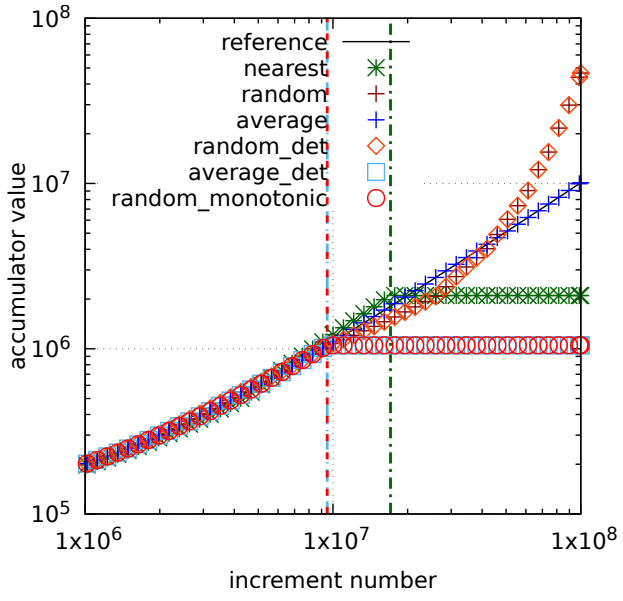
	Seq		Rec	
	float	double	float	double
error(nearest)	6.66	35.92	18.67	47.92
all	4.61	34.05	16.58	45.99
average	**6.67**	**35.91**	18.63	47.92
average_scomdet(xxhash)	**6.67**	**35.91**	**18.19**	**47.32**
average_scomdet(mersenne_twister)	6.67	35.91	**18.19**	**47.32**
sr_monotonic(xxhash)	**6.67**	**35.91**	**18.22**	**47.32**
sr_monotonic(mersenne_twister)	6.67	35.90	**18.25**	**47.32**

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

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

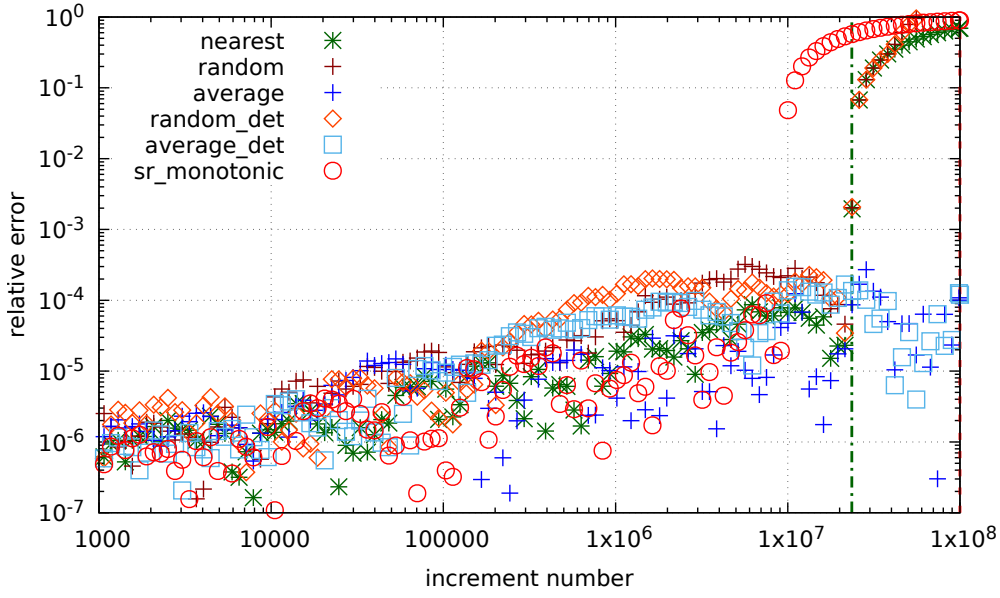
Effet sur la stagnation : incrément constant

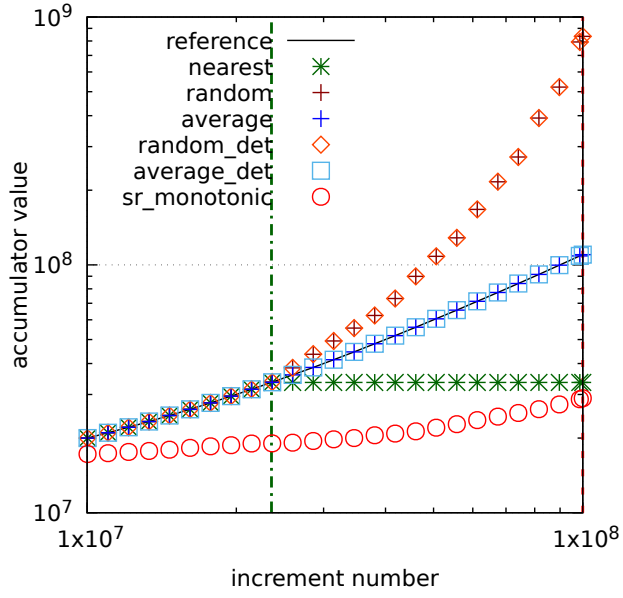




Error (and value) of accumulator (initialized to 100000) after i additions of 0.1.
The vertical bars represent the begining of stagnation.

Effet sur la stagnation : incrément random





Error (and value) of accumulator (initialized to 10000000) after i additions of random value (uniform distribution between 0 and 2)

Performances

Programme: stencil en float/double compilé en O0/O3 (avec fma)

type	double		float	
compilation option	O0	O3	O0	O3
nearest	x11.9	x24.9	x11.7	x32.4
random	x18.8	x52.2	x19.2	x77.6
random_det(xxhash)	x19.2	x51.7	x19.6	x80.5
random_comdet(xxhash)	x19.6	x53.7	x20.1	x83.4
random_scomdet(xxhash)	x23.8	x70.3	x25.0	x112.0
average	x21.7	x60.1	x22.2	x92.0
average_det(xxhash)	x23.4	x65.4	x24.5	x102.2
average_comdet(xxhash)	x24.6	x69.2	x26.3	x118.2
average_scomdet(xxhash)	x25.4	x72.4	x27.3	x124.8
sr_monotonic(xxhash)	x24.3	x70.7	x25.5	x110.7

Conclusions et perspectives

Les nouveaux modes `[random,average]_[[s]com]det` et `sr_monotonic`

- suppriment des faux-positifs sans besoin de modifier le code,
- simplifient le deboguage avec une graine fixée,
- ont un surcoût *acceptable* vis à vis de `[random,average]`,
- nécessitent de faire attention à la stagnation.

Perspectives

- Etude des applications utilisant `average` (IA typiquement) pour savoir dans quel régime elles sont vis-à-vis de la stagnation.
- Etude du mode `sr_monotonic` sur la convergence d'algorithme itératif.
- Besoin de REX sur les résultats du delta-debug avec les nouveaux modes stochastiques/déterministes.

Definition de prandom

Mode `random` :

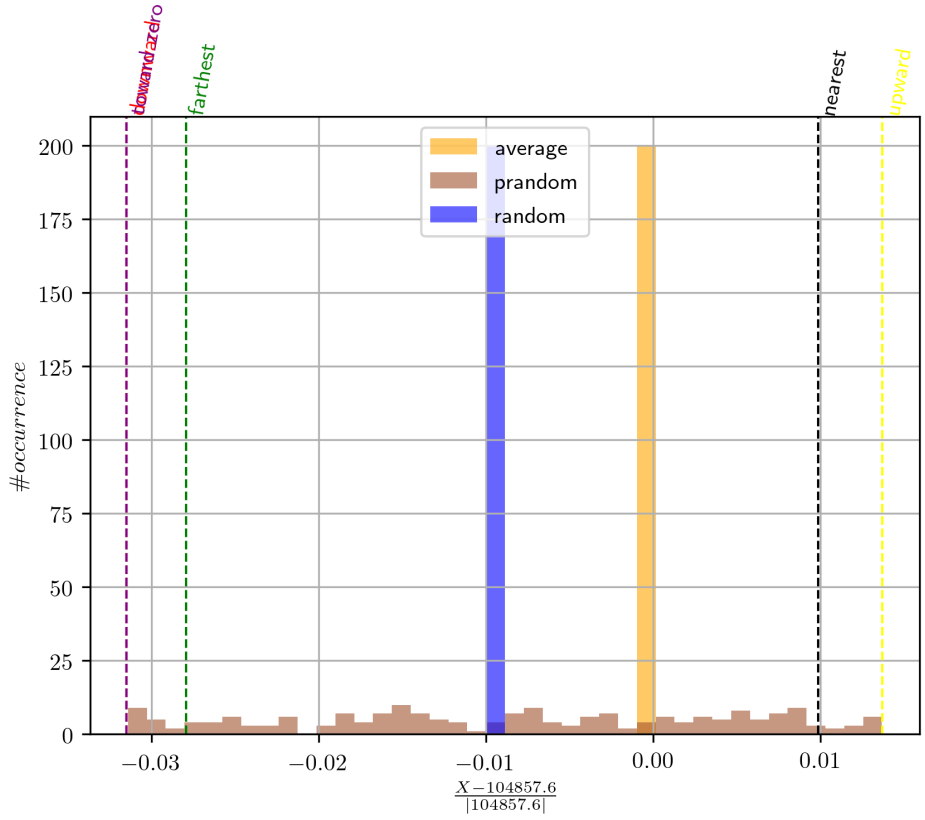
$$fl_{random}(a \circ b) = \begin{cases} \lfloor a \circ b \rfloor & avec\ p = \frac{1}{2} \\ \lceil a \circ b \rceil & avec\ p = \frac{1}{2} \end{cases}$$

Mode `prandom` :

$$fl_{prandom}(a \circ b) = \begin{cases} \lfloor a \circ b \rfloor & avec\ p = p_r \\ \lceil a \circ b \rceil & avec\ p = 1 - p_r \end{cases}$$

avec p_r selectionné suivant une loi uniforme entre 0 et 1 au début du programme.

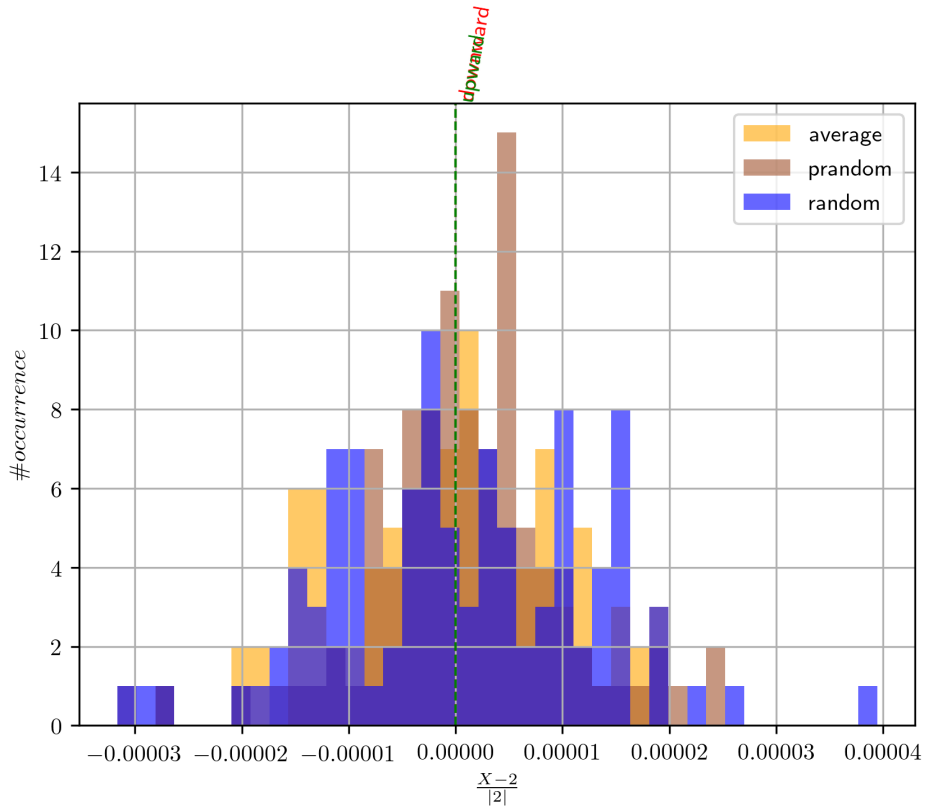
Exemple sur la somme



Exemple sur le conditionnement du produit scalaire

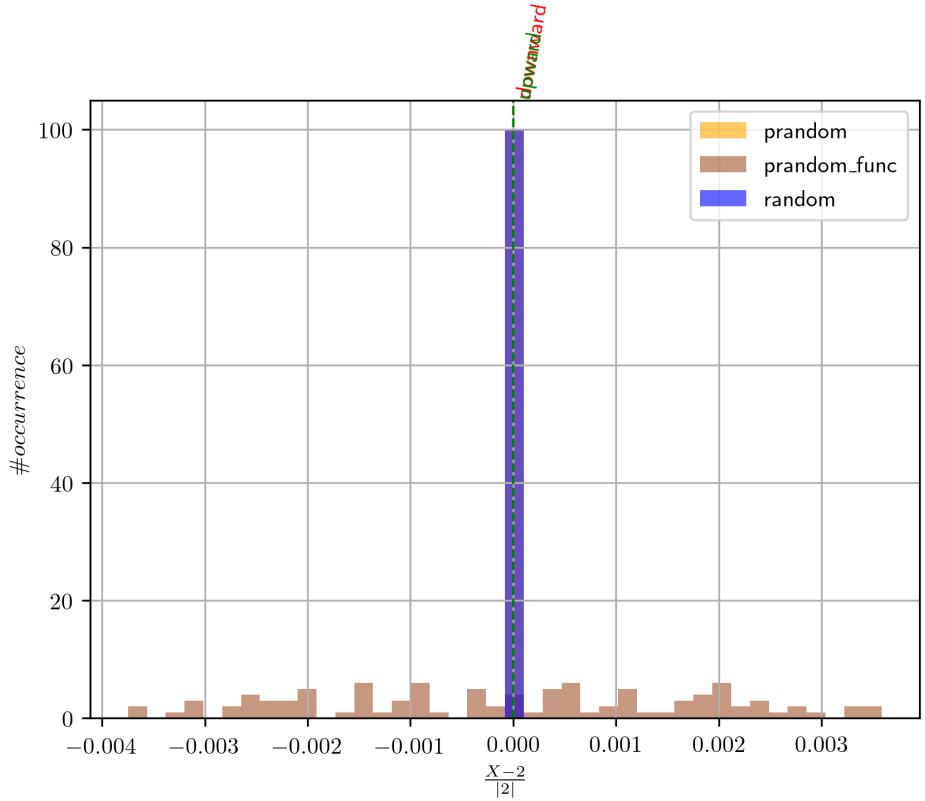
$$cond = 2.\frac{\sum_i |a_i.b_i|}{\sum_i a_i.b_i}$$

Avec $a_i > 0$ et $b_i > 0$, $cond = 2$



l'enjeu de l'update de p

Si on remet à jour p au début de chaque fonction :



Attention :

- Très sensible aux options de compilation et l'inlining.
- Si on remet à jour p à chaque opération prandom est équivalent à random

Perspectives sur prandom

Questionnement

- Où et comment placer les update?
- Pertinence sur des cas industriels pour la localisation par delta-debug?
- Pertinence sur des cas industriels pour la localisation par méthode de couverture?
- Pertinence sur des cas industriels pour la localisation par méthode d'analyse de donnée (cf. travail avec R.M.)?
- Version déterministe et update?