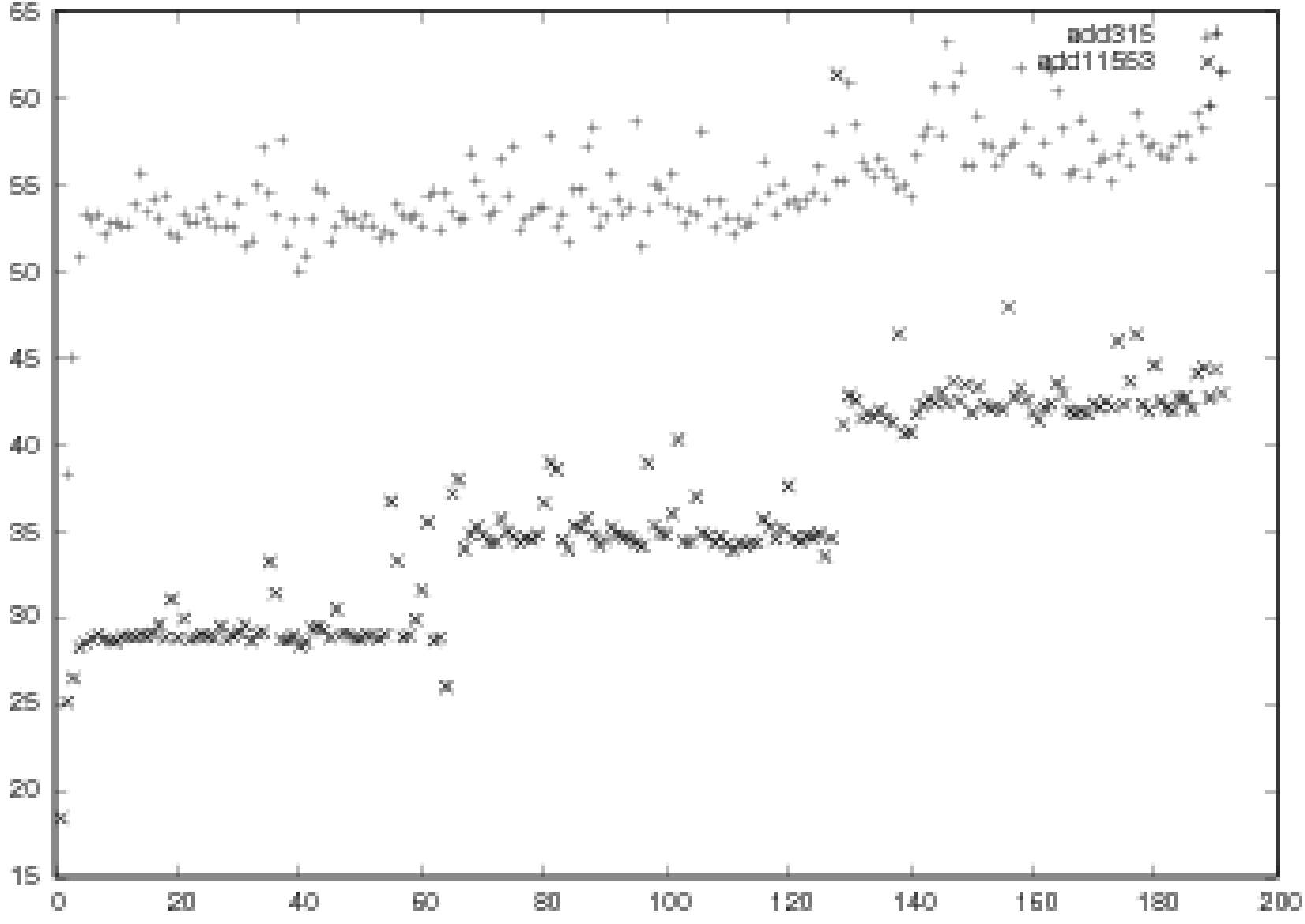
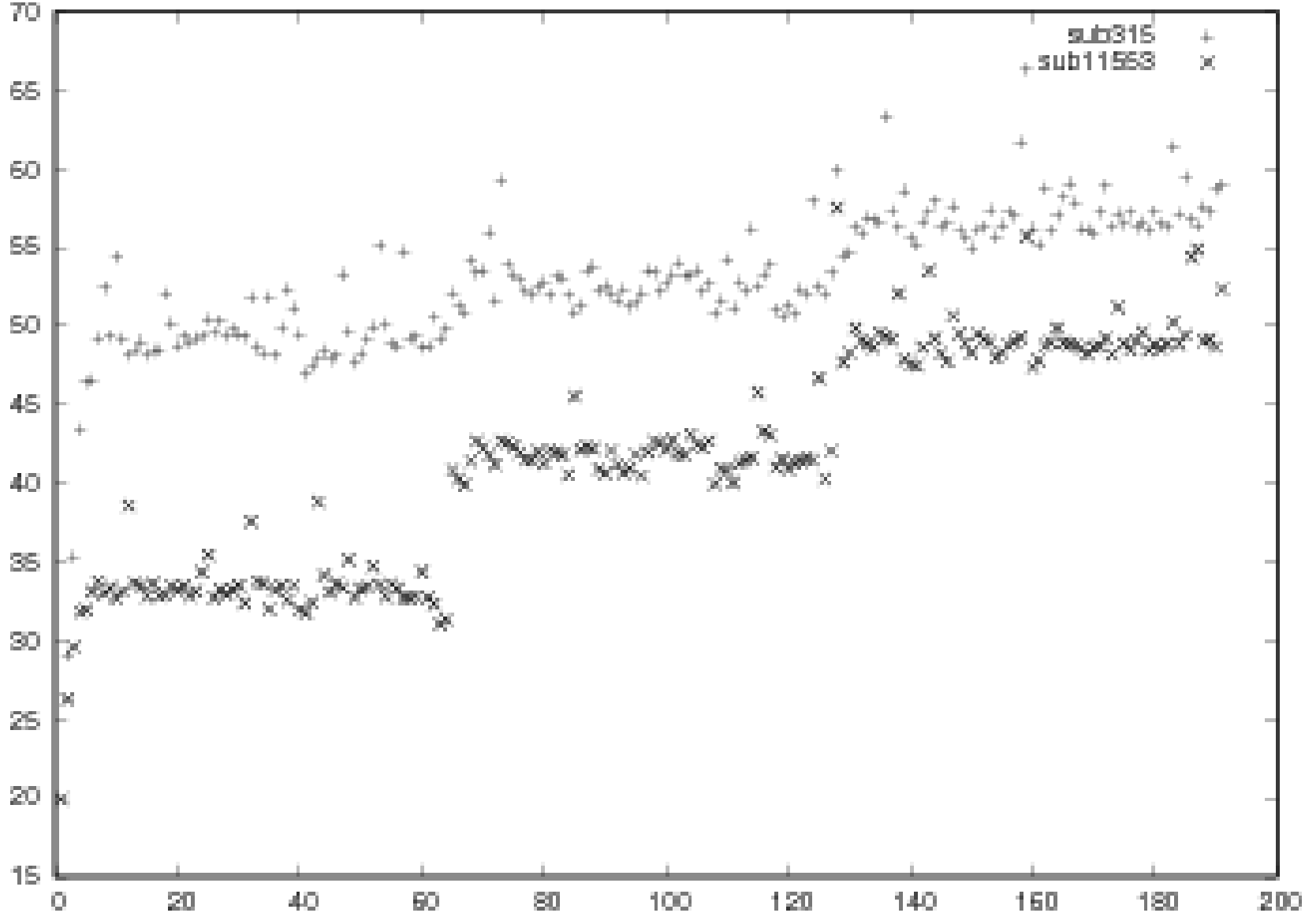
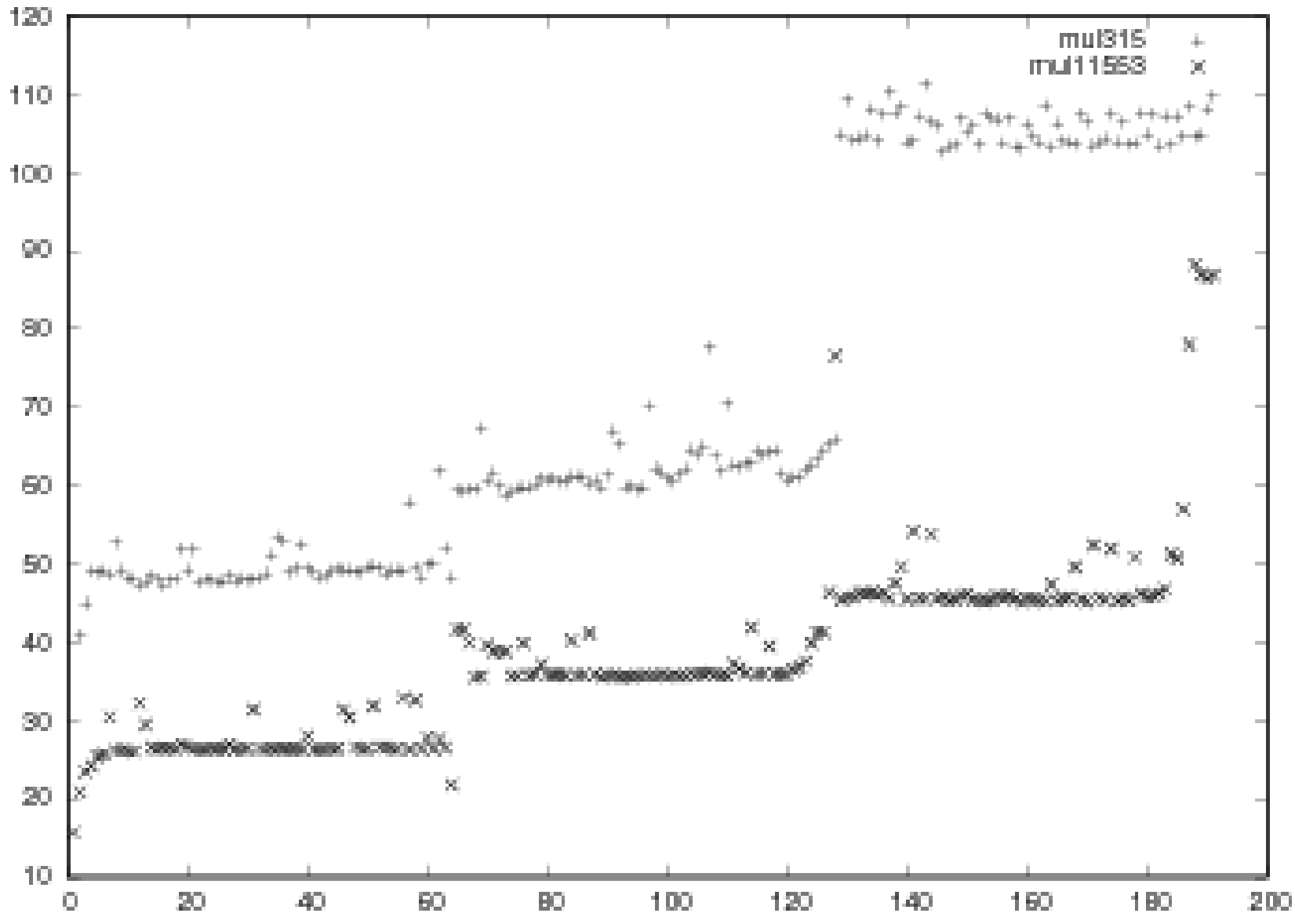
Des calculs plus rapides sur 1 et 2 mots-machine dans GNU MPFR (travail en commun avec Vincent Lefèvre)

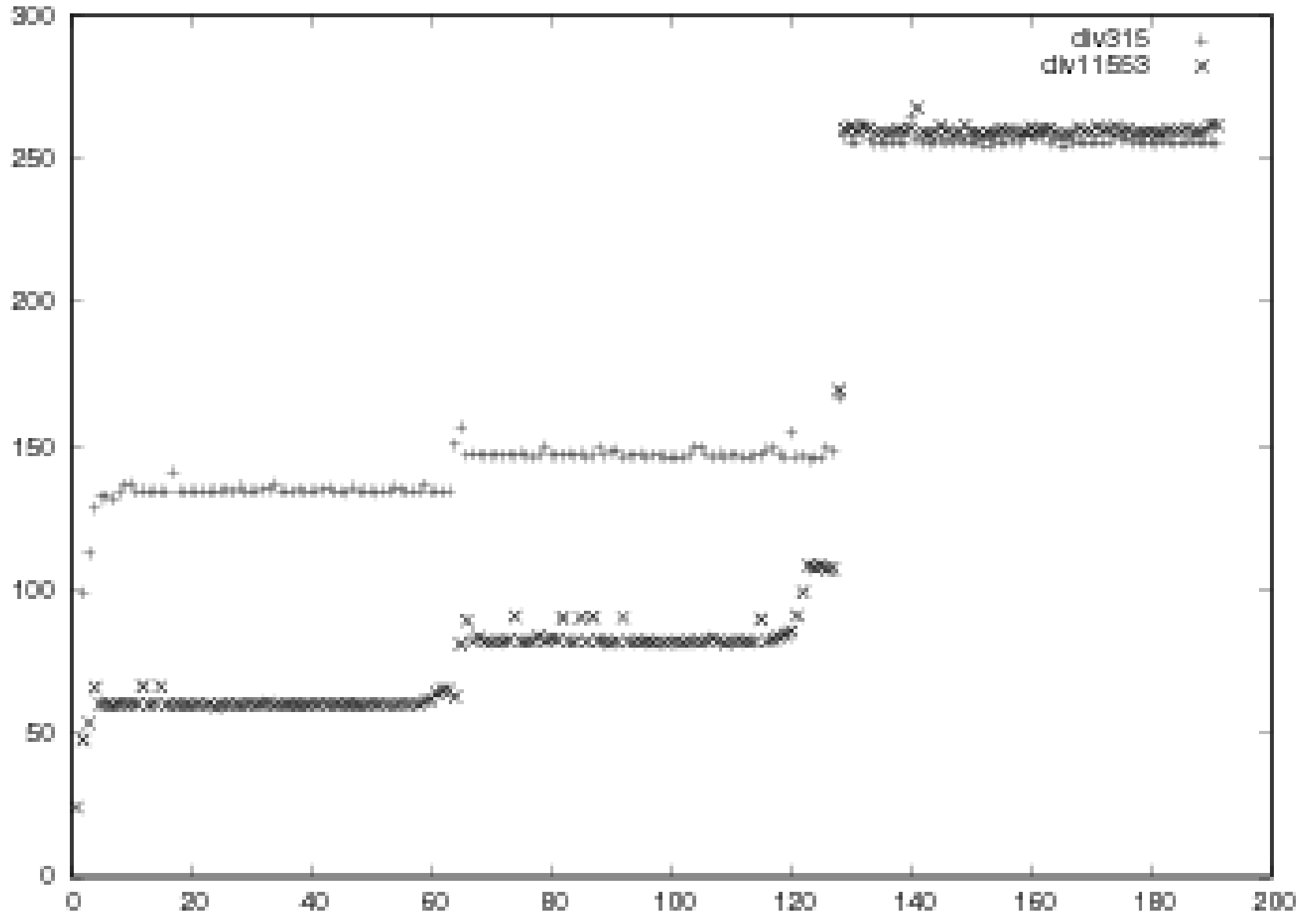
Paul Zimmermann

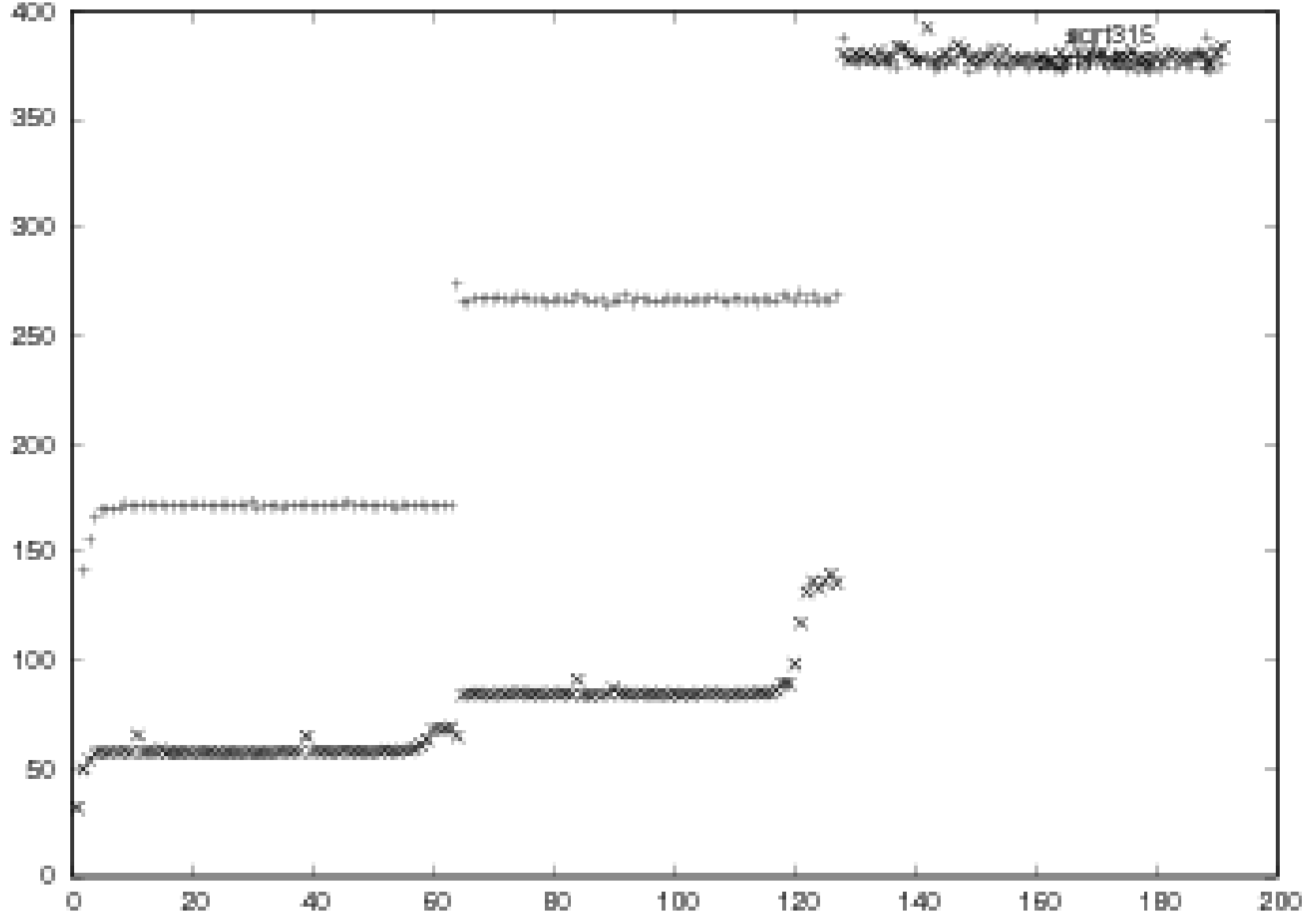
Journées FastRelax, Paris, 31 mai 2017


add315
add11563
65
60
55
50
45
40
35
30
25
20
15
0
20
40
60
80
100
120
140
160
180
200


sub315
sub11563
70
65
60
55
50
45
40
35
30
25
20
15
0
20
40
60
80
100
120
140
160
180
200


mul315
mul11563
120
110
100
90
80
70
60
50
40
30
20
10
0
20
40
60
80
100
120
140
160
180
200


div315
div11563
300
250
200
150
100
50
0
0
20
40
60
80
100
120
140
160
180
200


400
350
300
250
200
150
100
50
0
0
20
40
60
80
100
120
140
160
180
200
qgt315

Historique

- 1999 : l'idée de MPFR germe pendant l'Action de Recherche Collaborative « Calcul fiable »
- 2008 : MPFR est utilisé par GCC 4.3.0 (*constant folding*)

```
double x = sin (3.14);
```

- 2009 : MPFR devient GNU MPFR
- 2016 : 4e réunion des développeurs à Toulouse (en marge des journées FastRelax)

Arrondi correct

Principe : on calcule la valeur flottante la plus proche du résultat exact avec la précision et le mode d'arrondi donnés

RNDN : arrondi au plus proche

RNDZ : arrondi vers zéro, RNDA : arrondi en s'éloignant de zéro

RNDD : arrondi vers $-\infty$, RNDU : arrondi vers $+\infty$

Une seule valeur possible : c'est l'arrondi correct

Notations

MPFR utilise la couche `mpn` de GMP pour la représentation interne des mantisses

limb : un mot GMP (en général 32 ou 64 bits)

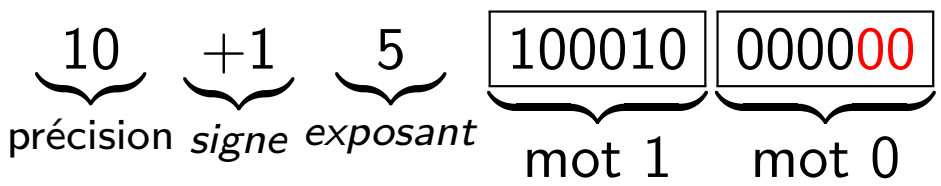
Pour simplifier on supposera qu'un *limb* fait 64 bits

Dans un mot de 64 bits, on appelle « bit 1 » le bit de poids fort, et « bit 64 » celui de poids faible.

Représentation des nombres MPFR (*mpfr_t*)

- précision $p \geq 1$ (en bits)
- signe (-1 ou $+1$)
- exposant (entre $E_{\min}$ et $E_{\max}$), utilisé aussi pour représenter les nombres spéciaux (NaN, $\pm\infty$, ± 0)
- mantisse (tableau de $\lceil p/64 \rceil$ *limbs*), définie seulement pour des nombres *reguliers* (pas NaN, ni $\pm\infty$, ni ± 0)

Les bits les plus significatifs sont notés à gauche.
Les nombres réguliers sont *normalisés* : le bit de poids fort du *limb* de poids fort doit être à 1.
Exemple, $x = 17$ avec une précision de 10 bits et des *limbs* de 6 bits est représenté ainsi :



Round bit et *sticky bit*

$$v = \underbrace{xxx\ldots yyy}_{m \text{ de } p \text{ bits}} \quad \underbrace{r}_{round\ bit} \quad \underbrace{sss\ldots}_{sticky\ bit}$$

Le *round bit* r est la valeur du bit $p+1$ (où le bit p est le dernier bit de la mantisse)

Le *sticky bit* s vaut zéro ssi `sss...` est nul.

Le *round bit* et le *sticky bit* suffisent à déterminer l'arrondi correct pour tous les modes d'arrondi :

r	s	vers zéro	au plus proche	*away*
0	0	m	m	m
0	1	m	m	$m+1$
1	0	m	$m+(m \bmod 2)$	$m+1$
1	1	m	$m+1$	$m+1$

La fonction mpfr_add

La fonction mpfr_add(a, b, c) fonctionne comme suit ($a \leftarrow b + c$) :

- tester en premier les valeurs singulières ($NaN, \pm Inf, \pm 0$)
- si b et c ont des signes différents, appeler mpfr_sub1
- si a, b, c ont la même précision, appeler mpfr_add1sp
- sinon appeler le code générique mpfr_add1 décrit dans : Vincent Lefèvre, *The Generic Multiple-Precision Floating-Point Addition With Exact Rounding (as in the MPFR Library)*, 6th Conference on Real Numbers and Computers 2004 - RNC 6, Nov 2004, Dagstuhl, Germany, pp.135-145, 2004.

La (nouvelle) fonction mpfr_add1sp

- si $p < 64$, appeler mpfr_add1sp1
- si $p = 64$, appeler mpfr_add1sp1n
- si $64 < p < 128$, appeler mpfr_add1sp2
- sinon exécuter le code générique pour des opérandes de même précision

Note : $p = 128$ utilise le code générique, il vaut mieux utiliser $p = 127$ si possible.

La fonction mpfr_add1sp1

Cas 1, $e_b = e_c$:

$$b = \boxed{110100}$$

$$c = \boxed{111000}$$

```
a0 = (bp[0] >> 1) + (cp[0] >> 1);
bx ++;
rb = a0 & (MPFR_LIMB_ONE << (sh - 1));
ap[0] = a0 ^ rb;
sb = 0;
```

Comme b et c sont normalisés, les bits de poids fort de $bp[0]$ et $cp[0]$ sont à 1.

Donc l'addition de $bp[0]$ et $cp[0]$ produit *toujours* une retenue, et l'exposant de a est $e_b + 1$ (ici `bx + 1`).

$$b = \boxed{110100}$$

$$c = \boxed{111000}$$

```
a0 = (bp[0] >> 1) + (cp[0] >> 1);
bx ++;
rb = a0 & (MPFR_LIMB_ONE << (sh - 1));
ap[0] = a0 ^ rb;
sb = 0;
```

La somme peut avoir $p + 1$ bits significatifs, mais comme $p < 64$ ($p < 6$ ici), elle tient en 64 bits.

`sh` est le nombre $64 - p$ de bits non utilisés, ici $6 - p = 2$.

Le *round bit* est le bit $p + 1$ de la somme, le *sticky bit* est toujours zéro.

On peut avoir un *overflow*, mais pas d'*underflow*.

La fonction mpfr_sub

La fonction mpfr_sub(a, b, c) fonctionne comme suit ($a \leftarrow b - c$) :

- tester en premier les valeurs singulières ($NaN, \pm Inf, \pm 0$)
- si b et c ont des signes différents, appeler mpfr_add1
- si b et c ont la même précision, appeler mpfr_sub1sp
- sinon appeler le code générique mpfr_sub1

La fonction mpfr_sub1sp

- si $p < 64$, appeler mpfr_sub1sp1
- si $p = 64$, appeler mpfr_sub1sp1n
- si $64 < p < 128$, appeler mpfr_sub1sp2
- sinon exécuter le code générique pour la soustraction avec des opérandes de même précision

Note : comme pour l'addition, il vaut mieux utiliser $p = 127$ que $p = 128$ si possible.

La fonction mpfr_sub1sp1

- si les exposants diffèrent, échanger b et c si besoin, de telle sorte que $e_b \geq e_c$
- cas 1 : $e_b = e_c$
- cas 2 : $e_b > e_c$

Cas 1, $e_b = e_c$:

$$b = \boxed{110100}$$

$$c = \boxed{111000}$$

calculer $bp[0] - cp[0]$ et stocker le résultat dans $ap[0]$, qui vaut $bp[0] - cp[0] \bmod 2^{64}$

si $ap[0] = 0$, le résultat est 0

si $ap[0] > bp[0]$, une retenue s'est produite, on a donc $|c| > |b|$: changer $ap[0]$ en $-ap[0]$ et changer le signe de a

sinon aucune retenue ne s'est produite, donc $|c| < |b|$

calculer le nombre k de zéros en tête de $ap[0]$, décaler $ap[0]$ de k bits vers la gauche et diminuer l'exposant de k

dans ce cas le *round bit* et le *sticky bit* sont toujours 0

On peut avoir un *underflow*, pas d'*overflow* : $|a| \leq \max(|b|, |c|)$.

La fonction mpfr_mul(a,b,c)

$$a \leftarrow \circ(b \cdot c)$$

- si $p_a = p_b = p_c < 64$, appeler mpfr_mul_1
- si $p_a = p_b = p_c = 64$, appeler mpfr_mul_1n
- si $64 < p_a = p_b = p_c < 128$, call mpfr_mul_2
- sinon utiliser le code générique

La fonction mpfr_mul_1

$$a \leftarrow \circ(b \cdot c)$$

a, b, c : au plus un *limb* (moins 1 bit)

$$h \cdot 2^{64} + \ell \leftarrow bp[0] \cdot cp[0] \qquad \text{(umul\_ppmm)}$$

Comme $2^{63} \leq bp[0], cp[0] < 2^{64}$, on a $2^{62} \leq h$

Si $h < 2^{63}$, décaler h, ℓ d'un bit vers la gauche, et diminuer l'exposant

Le *round bit* est le bit $p + 1$ de h ($p < 64$)

Le *sticky bit* est constitué des bits restants de h (aucun si $p = 63$) et ceux de ℓ

À la fois *underflow* et *overflow* peuvent arriver

Attention : MPFR considère l'*underflow après* l'arrondi (avec une plage d'exposant infinie)

La fonction mpfr_div(a,b,c)

$$a \leftarrow \circ(b/c)$$

- si $p_a = p_b = p_c < 64$, appeler mpfr_div_1
- si $p_a = p_b = p_c = 64$, appeler mpfr_div_1n
- si $64 < p_a = p_b = p_c < 128$, appeler mpfr_div_2
- sinon utiliser le code générique

La fonction mpfr_div_1

$$a \leftarrow \circ(b/c)$$

On a $p_a = p_b = p_c < 64$

1. $bp[0] \geq cp[0]$: un bit de quotient en plus
2. $bp[0] < cp[0]$: pas de bit de quotient en plus

Algorithme DivApprox1

Entrée : entiers u, v avec $0 \leq u < v$ et $\beta/2 \leq v < \beta$
Sortie : entier q approximation de $u\beta/v$

1: calculer un inverse approché i de v, vérifiant

$$i \leq \lfloor (\beta^2 - 1)/v \rfloor - \beta \leq i + 1$$

2: $q = \lfloor iu/\beta \rfloor + u$

Note : ici on a $\beta = 2^{64}$.

Le calcul de l'inverse approché est fait par une variante de la macro `invert_limb` de GMP (Möller et Granlund, *Improved division by invariant integers*, IEEE TC, 2011).

Théorème

Le quotient approché calculé par l'algorithme DivApprox1 vérifie

$$q \leq \lfloor \frac{u\beta}{v} \rfloor \leq q + 2$$

Conséquence : on peut déterminer l'arrondi exact de u/v, sauf si les derniers `sh-1` bits de q valent 000..000, 111..111 ou 111..110.

Dans ce cas (rare), pour accélérer le pire cas de l'algorithme, on repart de l'approximation q.

La fonction mpfr_sqrt(r,u)

$$r \leftarrow \circ(\sqrt{u})$$

- si $p_r = p_u < 64$, appeler mpfr_sqrt1
- si $p_r = p_u = 64$, appeler mpfr_sqrt1n
- si $64 < p_r = p_u < 128$, appeler mpfr_sqrt2
- sinon utiliser le code générique

Algorithme RecSqrtApprox1

Entrée : entier d avec $2^{62} \leq d < 2^{64}$
Sortie : entier v_3 approximation de $s = \lfloor 2^{96}/\sqrt{d} \rfloor$

1: $d_{10} = \lfloor 2^{-54} d \rfloor + 1$
2: $v_0 = \lfloor \sqrt{2^{30}/d_{10}} \rfloor$ (*table lookup*)
3: $d_{37} = \lfloor 2^{-27} d \rfloor + 1$
4: $e_0 = 2^{57} - v_0^2 d_{37}$
5: $v_1 = 2^{11} v_0 + \lfloor 2^{-47} v_0 e_0 \rfloor$
6: $e_1 = 2^{79} - v_1^2 d_{37}$
7: $v_2 = 2^{10} v_1 + \lfloor 2^{-70} v_1 e_1 \rfloor$
8: $e_2 = 2^{126} - v_2^2 d$
9: $v_3 = 2^{33} v_2 + \lfloor 2^{-94} v_2 e_2 \rfloor$

Remarque : si un *table lookup* est plus rapide qu'une multiplication, on peut tabuler v_0^2 à l'étape 4.

Théorème

La valeur v_3 renvoyée par RecSqrtApprox1 diffère d'au plus 8 de la racine carrée inverse :

$$v_3 \leq s := \lfloor 2^{96}/\sqrt{d} \rfloor \leq v_3 + 8.$$

Algorithme SqrtApprox1

Entrée : entier n avec $2^{62} \leq n < 2^{64}$

Sortie : entier r_0 approximation de $\sqrt{2^{64} n}$

1: calculer un entier x approximation de $2^{63}/\sqrt{n}$ avec

$$x \leq 2^{63}/\sqrt{n}$$

2: $y = \lfloor \sqrt{n} \rfloor$ (en utilisant l'approximation x)

3: $z = n - y^2$

4: $t = \lfloor 2^{-32} xz \rfloor$

5: $r_0 = y \cdot 2^{32} + t$

Théorème

Si l'approximation x à l'étape 1 est la valeur v_2 de l'algorithme RecSqrtApprox1, alors l'algorithme SqrtApprox1 renvoie r_0 tel que

$$r_0 \leq \lfloor \sqrt{2^{64} n} \rfloor \leq r_0 + 7.$$

La fonction mpfr_sqrt1

Entrée : $2^{63} \leq u < 2^{64}$ représentant un nombre de $p < 64$ bits (bit le plus significatif à 1)

- si l'exposant associé est impair, décaler u d'un bit vers la droite

Maintenant $2^{62} \leq u < 2^{64}$

- appeler `__gmpfr_sqrt_limb_approx`, qui implante SqrtApprox1, et calcule r_0 tel que

$$r_0 \leq \lfloor \sqrt{2^{64} u} \rfloor \leq r_0 + 7.$$

- si les `sh-1` bits de r_0 ne valent pas 000..000, 111..111 (-1), 111..110 (-2), 111..101 (-3), 111..100 (-4), 111..011 (-5), 111..010 (-6), 111..001 (-7), alors on sait déterminer l'arrondi correct.
- sinon on calcule $r = r_0 + i$ avec $0 \leq i \leq 7$ tel que

$$r = \lfloor \sqrt{2^{64} u} \rfloor$$

ce qui équivaut à :

$$0 \leq 2^{64} u - r^2 \leq 2r$$

MPFR 3.1.5 comparé à MPFR 4.0-dev

`araignee.loria.fr`, Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz, avec GMP 6.1.2 et GCC 6.3.0.
GMP et MPFR sont configurés avec `-disable-shared`

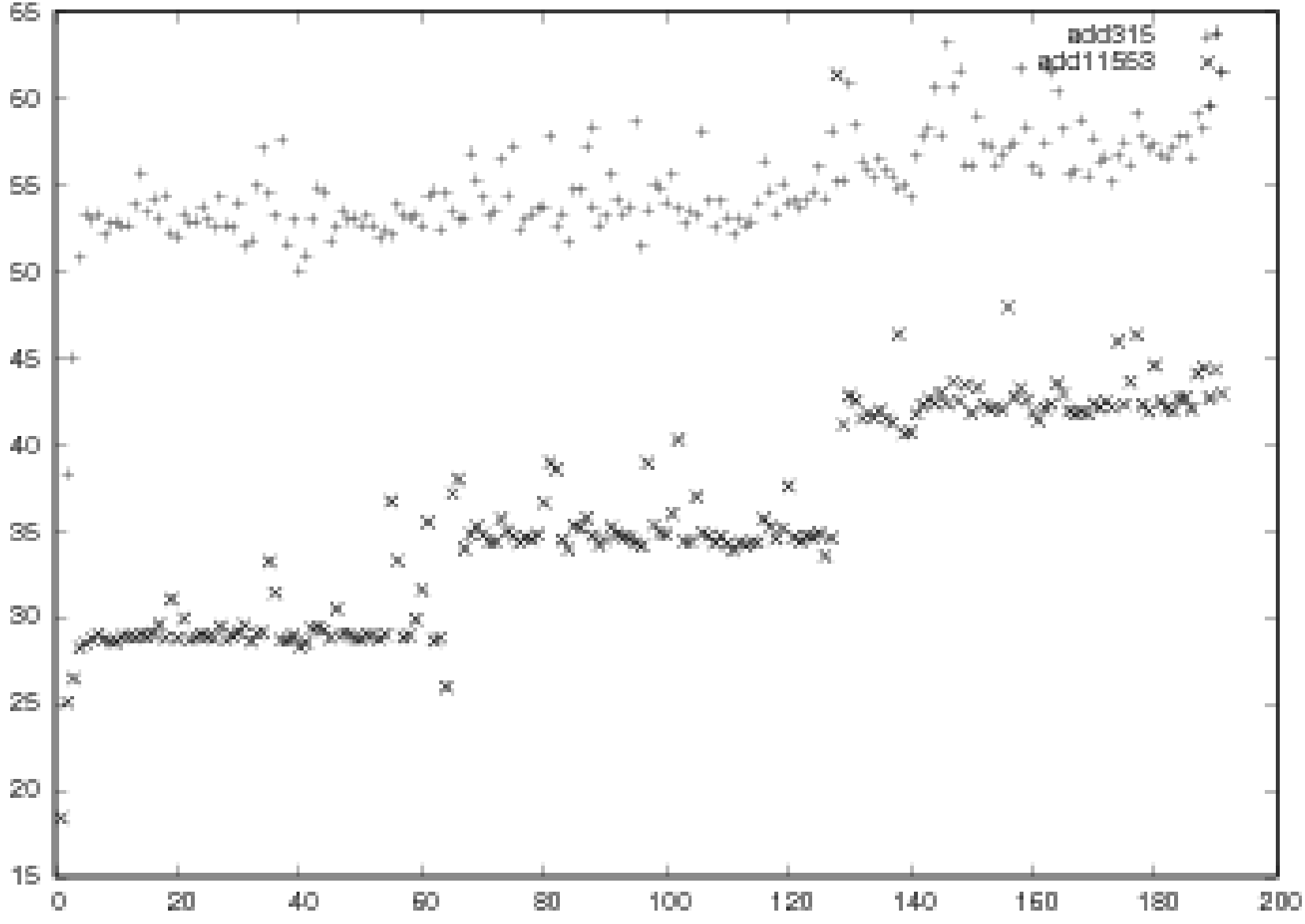
MPFR 3.1.5

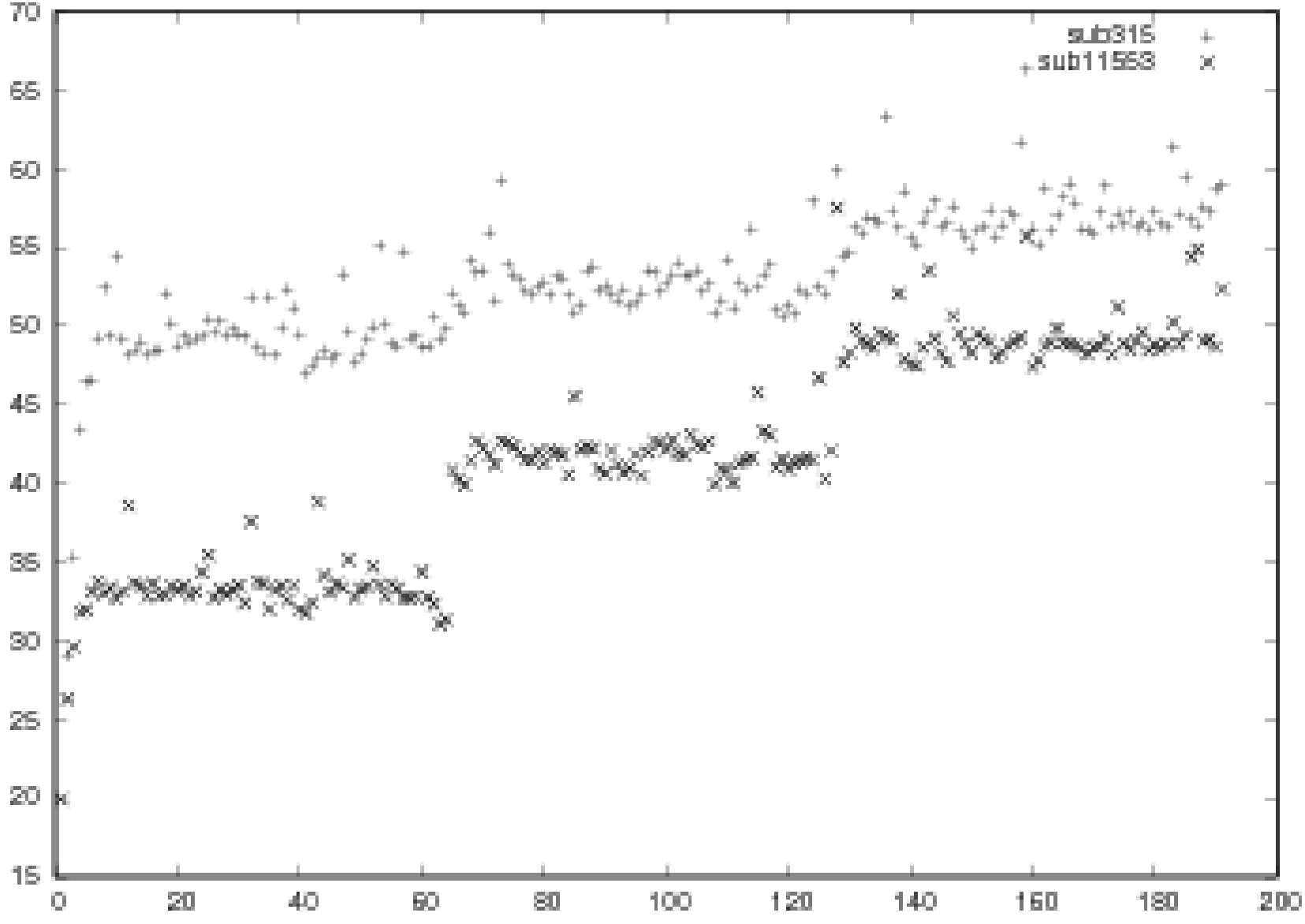
bits	53	113
mpfr_add	52	53
mpfr_sub	49	52
mpfr_mul	49	63
mpfr_sqr	74	79
mpfr_div	134	146
mpfr_sqrt	171	268

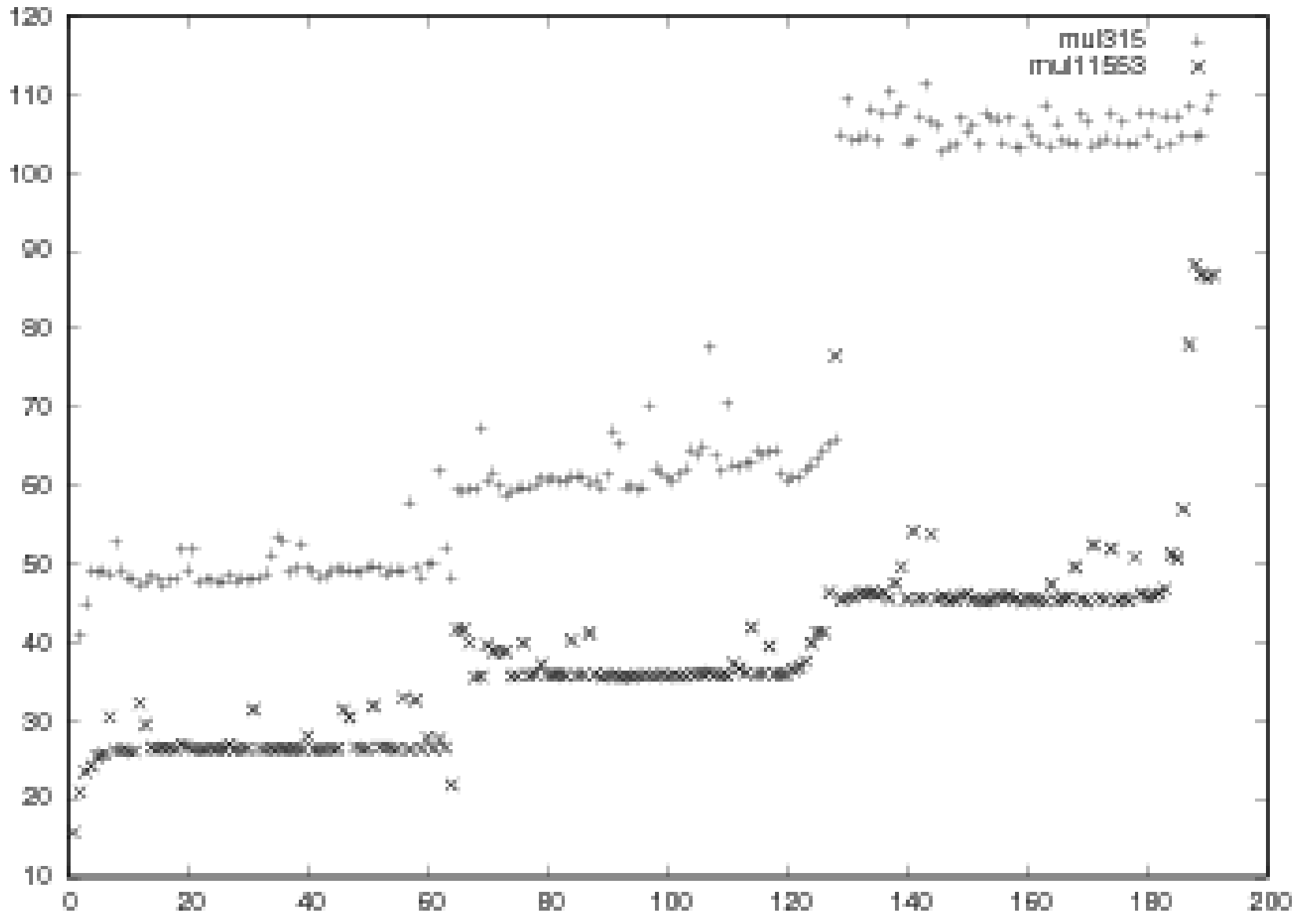
MPFR 4.0-dev

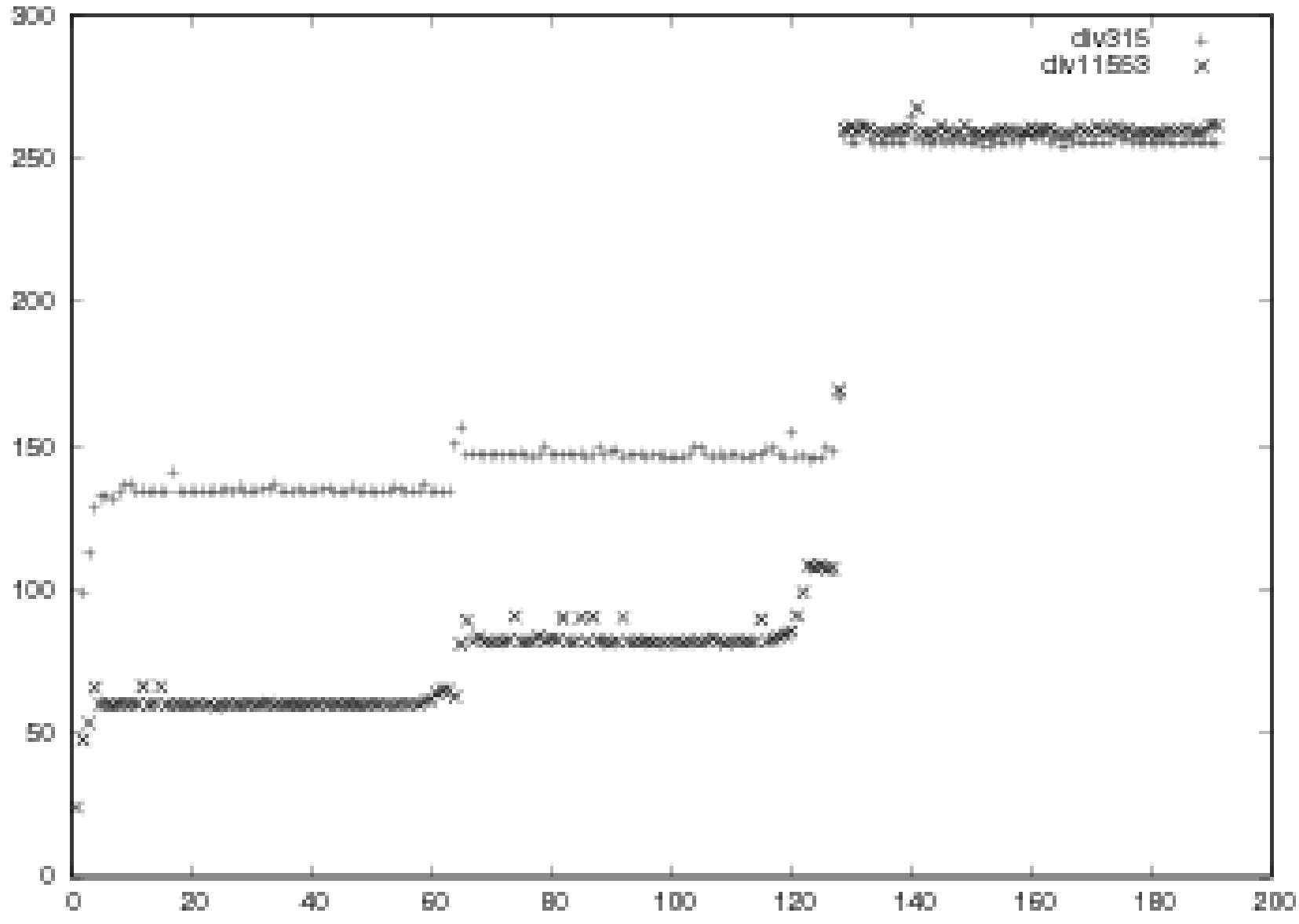
bits	53	113
mpfr_add	**25**	**29**
mpfr_sub	**28**	**33**
mpfr_mul	**23**	**33**
mpfr_sqr	**21**	**29**
mpfr_div	**56 (64)**	**77 (102)**
mpfr_sqrt	**55 (56)**	**84 (133)**

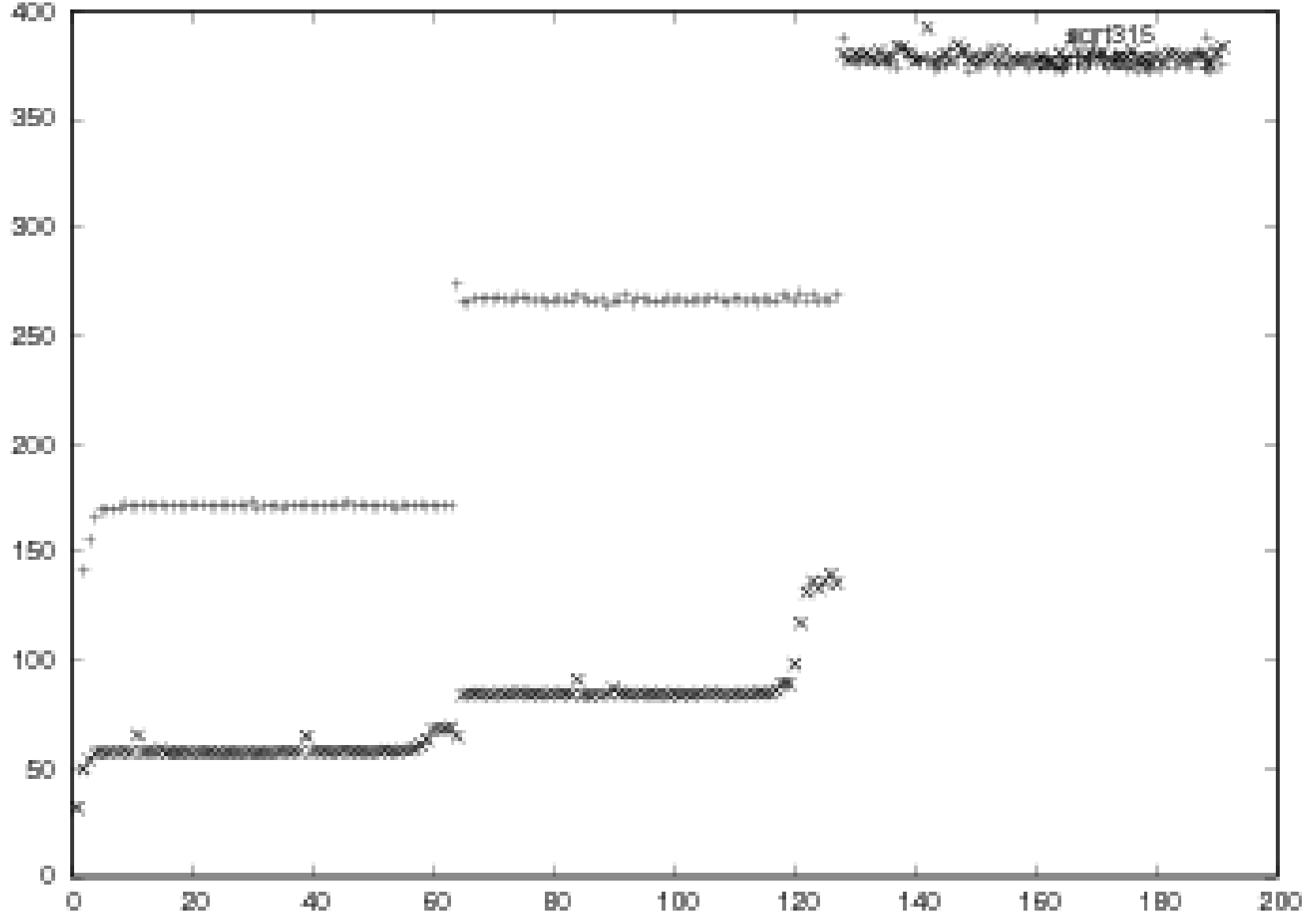
Les temps sont en cycles.


add315
add11563
65
60
55
50
45
40
35
30
25
20
15
0
20
40
60
80
100
120
140
160
180
200


sub315
sub11563
70
65
60
55
50
45
40
35
30
25
20
15
0
20
40
60
80
100
120
140
160
180
200


mul315
mul11563
120
110
100
90
80
70
60
50
40
30
20
10
0
20
40
60
80
100
120
140
160
180
200


div315
div11563
300
250
200
150
100
50
0
0
20
40
60
80
100
120
140
160
180
200


400
350
300
250
200
150
100
50
0
0
20
40
60
80
100
120
140
160
180
200
qgt315

Conclusion

Gain d'un facteur 2 ou plus jusque 127 bits pour $\div, \sqrt{}$, jusque 191 bits pour $+, -, \times$.

À venir dans MPFR 4, déjà disponible dans la version de développement !

De nouveaux algorithmes *fast* and *relax(ed)* pour la division et la racine carrée, avec des bornes petites et fines.

Du code et des preuves bien explicités, prêts pour une preuve formelle. Des volontaires pour trouver un bug ?