École CEA-EDF-INRIA Calcul numérique certifié

Travaux pratiques: Premiers pas avec MPFR

25-26 octobre 2007

1 Prise en main de MPFR

Problème 1.1 *Écrire un programme calculant* $\exp(1.5)$ *avec 150 bits de précision et affichant le résultat en décimal en arrondi au plus près. Augmentez la précision à 170 bits. Que remarquez-vous ?*

Faites varier la précision de calcul de 130 à 150 bits et affichez à chaque fois le résultat sur 41 chiffres décimaux en arrondi au plus près.

Problème 1.2 *Écrire un programme évaluant le polynôme de Rump :*

$$333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}$$

en $a = 77617$ *et* $b = 33096$ *dans toutes les précisions de 17 à 128 bits (sans réordonner les termes), et affichant, pour chaque précision, le résultat en décimal en arrondi au plus près.*

Problème 1.3 *Écrire un programme calculant* 1.001^n *pour* n *allant de* 2 *à* 9 *et affichant le résultat sur 28 chiffres décimaux.*

Problème 1.4 *Écrire un programme calculant* $\sqrt[3]{7 + \sqrt[5]{2} - 5\sqrt[5]{8}} + \sqrt[5]{4} - \sqrt[5]{2}$ *en utilisant le mode d'arrondi au plus près. Votre programme pourra prendre en argument la précision des calculs intermédiaires.*

Problème 1.5 *Écrire un programme calculant* $\sqrt{\sin^2 x + \cos^2 x}$ *en utilisant le mode d'arrondi au plus près (il vous est juste interdit de simplifier mathématiquement). Votre programme prendra en argument la valeur de* x *et la précision des calculs intermédiaires. Testez votre programme sur diverses valeurs de* x*, y compris de grosses valeurs comme* 10^{17}*. Que remarquez-vous ?*

Problème 1.6 *Écrire un programme calculant e à l'aide de la formule* $e = \sum_{k=0}^{\infty} 1/k!$ *avec une erreur aussi petite que possible en moins en 2 secondes (cf annexe). Votre programme pourra prendre en argument un ou plusieurs paramètres (e.g. précision). Après avoir estimé l'erreur, vous pourrez comparer votre résultat avec celui de* `mpfr_exp` *et afficher une borne sur votre erreur, ainsi que le temps de calcul.*

Problème 1.7 *Écrire un programme calculant les termes de la suite :*

$$\begin{cases} u_0 &= 2 \\ u_1 &= -4 \\ u_{n+1} &= 111 - \dfrac{1130}{u_n} + \dfrac{3000}{u_n u_{n-1}} \end{cases}$$

en utilisant le mode d'arrondi au plus près. Votre programme pourra prendre en argument : une précision globale et le nombre de termes à afficher. Que remarquez-vous ? Quelles sont les 11 premières décimales de u_{100} *?*

Problème 1.8 *Écrire un programme calculant les termes de la suite :*

$$\begin{cases} u_0 &= e - 1 \\ u_n &= n.u_{n-1} - 1 \end{cases}$$

en utilisant le mode d'arrondi au plus près. Quelles sont les 22 premières décimales de u_{998} *en utilisant une précision de 8586 bits pour tous les calculs intermédiaires ? Quelles sont les 19 premières décimales de* u_{998} *?*

Problème 1.9 *Reprendre les problèmes précédents en utilisant les modes d'arrondi dirigés pour fournir un encadrement du résultat exact.*

Problème 1.10 *Utilisez la méthode de Newton sur* $x^2 - x - 1 = 0$ *en partant de* $x = 1$ *pour calculer approximativement n bits du nombre d'or* φ*. Vérifier le résultat à l'aide de la formule* $\varphi = \frac{1+\sqrt{5}}{2}$ *; vous pourrez afficher la différence en ulp.*

2 Arrondi correct

Nous allons proposer deux méthodes pour implémenter une fonction simple avec arrondi correct : $f_n(x) = 1/\sqrt[n]{x} = x^{-1/n}$, où x est un nombre MPFR et n un entier de type `unsigned long`. Dans les deux cas, il faudra utiliser la stratégie de Ziv. Afin de pouvoir vérifier, vous calculerez ainsi $1717^{-1/17}$ avec 692 bits de précision en arrondi au plus près, et 693 dans les modes d'arrondi dirigés. Vous afficherez le résultat en base 2, sur 692 (resp. 693) chiffres.

Problème 2.1 *La fonction f_n sera implémentée avec arrondi correct par composition des deux fonctions $g_n(x) = \sqrt[n]{x}$ et $h(x) = 1/x$. Laquelle de ces deux fonctions g_n et h vaut-il mieux appliquer en premier ? Pourquoi ?*

Problème 2.2 *Implémenter la fonction f_n avec arrondi correct. Vous ne ferez ici pas d'analyse d'erreur : vous utiliserez simplement la monotonie des fonctions g_n et h, en faisant le calcul deux fois et en jouant sur les modes d'arrondi (en vous inspirant de l'arithmétique d'intervalles).*

Problème 2.3 *Implémenter la fonction f_n avec arrondi correct en utilisant la fonction* `mpfr_can_round` *décrite dans la section* Internals *du manuel de MPFR.*

Problème 2.4 *Modifier le code écrit pour chacune des deux méthodes de façon à retourner une valeur ternaire comme le fait MPFR. Quelles valeurs ternaires obtenez-vous sur l'exemple demandé ?*

3 Exceptions

Le mécanisme d'exceptions (implémenté par MPFR à l'aide de *flags* globaux) est utile pour écrire du code générique et faire une vérification *a posteriori* pour traiter les cas exceptionnels en réécrivant des expressions sous une autre forme.

Une fonction f est avec arrondi fidèle si pour tout x, l'évaluation de $f(x)$ retourne la valeur exacte $f(x)$ arrondie soit vers $-\infty$, soit vers $+\infty$.

Problème 3.1 *Implémenter la fonction $\sqrt{x+y}$ avec arrondi fidèle. Vous la testerez dans les précisions 32 bits et 128 bits en affichant le résultat en base 16, sur les exemples suivants, et en intervertissant x et y :*

x	y
144	145
2^{62}	$2^{32}+2$
$2^{emax}(1-2^{-34})$	17
$2^{emax}(1-2^{-34})$	2^{emax-3}
$2^{emin}(1-2^{-34})$	$-2^{emin}(1-2^{-17})$
$+\infty$	1
$+\infty$	$+\infty$

Problème 3.2 *Implémenter la fonction $\log(\exp(x)/17-2)$ avec arrondi fidèle. Vous la testerez dans les précisions 32 bits et 128 bits en affichant le résultat en base 16, sur les exemples suivants : 4, 42, $-\infty$, $+\infty$, ± 0, 2^{32}, $2^{emax}(1-2^{-34})$, $\log 34$ et $\log 51$ tous deux arrondis sur 512 bits vers $-\infty$ et vers $+\infty$.*

4 Émulation d'autres arithmétiques

Les fonctions `mpfr_set_emin`, `mpfr_set_emax` et `mpfr_subnormalize` peuvent servir à émuler des arithmétiques du style IEEE 754. La fonction `mpfr_check_range` peut également être utile.

Problème 4.1 *Implémenter* `void poly(mpfr_t y, mpfr_t x, int d, mpfr_t *a)` *qui évalue le polynôme* $y = P(x) = \sum_{i=0}^{d} a_i x^i$ *par le schéma de Horner, comme si vous aviez une arithmétique IEEE 754 en double précision. Bien entendu, on suppose que les valeurs* x *et* a_i *sont représentables en double précision IEEE.*

Problème 4.2 *Émuler les arithmétiques suivantes sur le code* `floor(x/y)` *(forme très souvent utilisée pour effectuer une division euclidienne lorsque l'on travaille sur des types flottants) :*

- *arithmétique IEEE 754 double précision ;*
- *idem avec précision intermédiaire étendue x86 (64 bits de mantisse), i.e. avec un double arrondi : d'abord sur 64 bits, puis sur 53 bits.*

Tester le code sur des exemples simples, ainsi que sur $x = 3 \times 2^{52} - 4$ *et* $y = 2^{52} - 1$.

Problème 4.3 *Reprendre le problème 4.1 pour cette fois implémenter une évaluation polynomiale avec arrondi correct pour le format double IEEE.*

Bien que la conversion base 2 – base 10 soit avec arrondi correct, le résultat d'un simple calcul n'est pas forcément l'arrondi correct du résultat mathématique, car MPFR calcule en base 2 (phénomène du double arrondi).

Problème 4.4 *Pour* p *et* q *entiers de type* `unsigned long`*, écrire un programme qui affiche la valeur de* p/q *correctement arrondie au plus près sur 6 chiffres en décimal. Y a-t-il des valeurs qui peuvent poser problème ? Tester sur* $500\,002\,499$ / $499\,999\,999$.

5 Annexe : mesure du temps de calcul

Le temps de calcul (à la différence du temps réel) peut se mesurer avec la fonction `clock()`, déjà présente dans C89. Exemple :

```
#include <stdio.h>
#include <time.h>

int main (void)
{
  clock_t c;
  volatile int i;

  c = clock ();
  for (i = 0; i < 1000000000; i++)
    { }
  c = clock () - c;
  printf ("%g seconds\n", (double) c / CLOCKS_PER_SEC);
  return 0;
}
```

Note : sous Linux, tapez `man 7 time` pour avoir les différentes manières de mesurer le temps. Pour les calculs longs sur systèmes POSIX, il est préférable d'utiliser la fonction POSIX `times()` car sur les machines 32 bits, `clock()` reprend la même valeur au bout de 72 minutes environ.