TP de GNU MPFR

Vincent Lefèvre

26 mars 2013

1 Introduction

GNU MPFR [1] est une bibliothèque de calcul à virgule flottante en multiprécision (en base 2), s'inspirant des bonnes idées de la norme IEEE 754 (arrondi correct, nombres spéciaux, exceptions...). Les calculs sont à la fois efficaces et bien définis sémantiquement. MPFR est basée sur la bibliothèque GMP.

1.1 Introduction à GMP

GMP [2] (GNU MP) est une bibliothèque libre de calcul en multiprécision, focalisée sur la performance.

Attention ! Pour des questions de performance justement, lors de la compilation de GMP, une ABI différente de celle utilisée par le système (compilateur et bibliothèques) peut être sélectionnée par GMP. Conséquence : sur certaines plateformes, pour compiler vos programmes utilisant GMP, l'option `-m64` peut être nécessaire.

La brique de base est le *limb* (entier non signé) représentant un chiffre dans une grande base (e.g. 2^{32} ou 2^{64}). La bibliothèque GMP se compose de différentes couches (en plus de fonctions génériques, comme les fonctions d'entrées/sorties formatées) :

- MPN (préfixe `mpn_`) : couche de bas niveau. Opérations sur des tableaux de *limb*s, représentant des entiers naturels.
- MPZ (préfixe `mpz_`, type `mpz_t`) : arithmétique entière signée.
- MPQ (préfixe `mpq_`, type `mpq_t`) : arithmétique rationnelle.
- MPF (préfixe `mpf_`, type `mpf_t`) : arithmétique flottante basique (pas de fonctions élémentaires, pas de gestion d'exceptions, peu de contrôle de la précision). Couche obsolète : utiliser MPFR.

Ce qui suit concerne essentiellement les couches de haut niveau, mais les conventions d'appel des fonctions de la couche MPN sont similaires.

Chaque type de base (e.g. `mpz_t`) est défini comme un tableau à un élément d'une certaine structure, ce qui signifie que lors du passage d'une donnée d'un tel type à une fonction, c'est juste le *pointeur* qui est passé. Les appels de fonction (opérations) ont alors la forme suivante. Par exemple, pour l'addition $a = b + c$ de deux entiers a et b en multiprécision :

```
mpz_add (a, b, c);
```

où chaque variable est de type `mpz_t`. La convention est que le premier argument d'une fonction contiendra le résultat. Certaines fonctions peuvent prendre en argument un entier natif (`long` ou `unsigned long`).

1. `http://www.mpfr.org/`
2. `http://gmplib.org/`

Exemples de prototypes :

```
void mpz_add_ui (mpz_t a, mpz_t b, unsigned long c)
void mpz_ui_sub (mpz_t a, unsigned long b, mpz_t c)
void mpz_mul_si (mpz_t a, mpz_t b, long c)
```

L'affectation se fait avec `mpz_set`, `mpz_set_ui`, `mpz_set_si` ou `mpz_set_str`.

Avant toute utilisation, une variable `x` doit être initialisée (une seule fois), par `mpz_init(x)` en général. On peut combiner initialisation et affectation : `mpz_init_set`, `mpz_init_set_ui`, etc. Pour libérer l'espace mémoire pris par une variable initialisée : `mpz_clear(x)`.

1.2 Introduction à MPFR

Les conventions s'inspirent de GMP : le préfixe est `mpfr_` et le type de base est `mpfr_t`. Les principales différences avec GMP : à l'appel d'une fonction, un mode d'arrondi est fourni comme dernier argument (`MPFR_RNDN`, `MPFR_RNDD`, `MPFR_RNDU` ou `MPFR_RNDZ`) et la plupart des fonctions renvoient une valeur entière, dont le signe indique le signe de l'erreur (en particulier, l'entier renvoyé est nul ssi le résultat est exactement représentable).

Contrairement à GMP, MPFR permet d'initialiser ou de libérer plusieurs variables à la fois (pour l'initialisation, les variables doivent avoir la même précision initiale) :

```
#define VARS x, y, z
mpfr_t VARS;
mpfr_inits2 (prec, VARS, (mpfr_ptr) 0);
/* ... */
mpfr_clears (VARS, (mpfr_ptr) 0);
```

Exemple de code :

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

int main (void)
{
  unsigned long i;
  mpfr_t s, t, u;

  mpfr_init2 (t, 200); mpfr_set_d (t, 1.0, MPFR_RNDD);
  mpfr_init2 (s, 200); mpfr_set_d (s, 1.0, MPFR_RNDD);
  mpfr_init2 (u, 200);
  for (i = 1; i <= 100; i++)
    { mpfr_mul_ui (t, t, i, MPFR_RNDU);
      mpfr_set_ui (u, 1, MPFR_RNDD);
      mpfr_div (u, u, t, MPFR_RNDD);
      mpfr_add (s, s, u, MPFR_RNDD); }
  mpfr_printf ("Sum is %RDe\n", s);  /* D: rounding mode */
  /* mpfr_out_str (stdout, 10, 0, s, MPFR_RNDD); putchar ('\n'); */
  mpfr_clears (s, t, u, (mpfr_ptr) 0);
  return 0;
}
```

Compiler avec `-lgmp -lmpfr`.

2 Prise en main de MPFR

Exercice 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.

Exercice 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.*

Exercice 3 *Écrire un programme calculant* 1.001^n *(précisément) pour* n *allant de* 2 *à* 9 *et affichant le résultat sur 28 chiffres décimaux.*

Exercice 4 *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.*

Exercice 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^{500000}*. Que remarquez-vous ?*

Exercice 6 *Écrire un programme calculant les termes de la suite (JMM1) :*

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

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} *?*

Exercice 7 *Écrire un programme calculant les termes de la suite (JMM2) :*

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

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} *?*

3 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.