Développement en fraction continue

Le problème inverse du développement décimal d'un rationnel consiste à trouver des rationnels qui approchent le mieux possible un réel donné ; par exemple :

22 / 7 = 3.1428... et 355 / 113 = 3.1415929...

sont de bonnes approximations (surtout la seconde) de *pi*=3.1415926... Pour découvrir de bonnes approximations rationnelles de y>0, on développe y en fraction continue, c'est à dire sous la forme :

```
                   1
y = x0 + ---------------
                     1
         x1 + ----------
                        1
              x2 + -----
                      ...
```

où x_0 est la partie entière de y, x_1 la partie entière de $1/(y-x_0)$, et ainsi de suite. Par exemple :

```
                 1                  1                     1
pi = 3 + --------- = 3 + ------------- = 3 + -----------------
         7.0625...                   1                     1
                         7 + ---------       7 + -------------
                             15.996...                      1
                                                 15 + --------
                                                      1.003...
```

Tout rationnel possède un développement en fraction continue fini (la réciproque est évidente) : en effet, si $y=a/b$, et si on effectue la division euclidienne :

$$a = b\,q + r$$

alors $x_0 = q$, et on poursuit le développement en fraction continue en remplaçant a/b par b/r : c'est l'algorithme d'Euclide.

Tout réel est représenté approximativement en machine par le rationnel :

$$2^e\,(1 + m_1 / 2 + m_2 / 4 + m_3 / 8\ \ldots)$$

où e est l'exposant et les m_i les chiffres de la mantisse. Donc le développement en fraction continue devrait être fini ; mais les calculs sont effectués de façon approchée, et les erreurs d'arrondi s'accumulent rapidement. Il faut donc borner l'ordre du développement.

Voici une fonction qui écrit le développement en fraction continue d'ordre au plus n d'un réel x dans le tableau t ; elle retourne la longueur du développement.

```
#include <math.h>

int fraction_continue (long t[], double x, int n) {
  int i;
  double delta, epsilon = 1e-8;

  for (i = 0; i < n; i++) {
    t[i] = (long) x;
    delta = x - t[i];
    if ( fabs (delta) < epsilon )
      return i + 1;
    x = 1 / delta;
  }
  return i;
```

```
    }
```

Commentaires de programmation

- L'expression `(long) x` réalise une conversion de type : x est de type `double`, et le résultat de type `long`, avec troncature des décimales.
- A cause des erreurs d'arrondi, le test mathématique `delta = 0` est remplacé par le test `|delta| < epsilon`.

On appelle développement partiel (ou *réduite*) de rang n la fraction :

$$\frac{a_n}{b_n} = x_0 + \cfrac{1}{x_1 + \cfrac{1}{\dots + \cfrac{1}{x_n}}}$$

Par exemple, les réduites de rang 1, 2, 3 du développement en fraction continue de *pi* sont :

$$3 + \frac{1}{7} = \frac{22}{7}\ ,\quad 3 + \cfrac{1}{7 + \cfrac{1}{15}} = \frac{333}{106}\ ,\quad 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1}}} = \frac{355}{113}$$

On démontre que toute réduite a_n / b_n de y est la meilleure approximation rationnelle possible de y, parmi les nombres de dénominateur inférieur ou égal à b_n. Le numérateur a_n et le dénominateur b_n se calculent facilement en lisant le développement de droite à gauche, et de bas en haut; on démontre que ce calcul fournit directement la fraction sous forme *irréductible*. Plus précisément, soit :

$$\frac{p_k}{q_k} = x_k + \cfrac{1}{x_{k+1} + \cfrac{1}{\dots + \cfrac{1}{x_n}}}$$

On a :

$$\frac{p_k}{q_k} = x_k + \frac{1}{p_{k+1} / q_{k+1}}$$

d'où :

$$p_k = p_{k+1}\, x_k + q_{k+1}$$
$$q_k = p_{k+1}$$

Avec les conditions initiales $p_n = x_n$, $q_n = 1$, ces formules permettent de calculer la réduite

$a_n / b_n = p_0 / q_0$ (la récurrence semble fonctionner à l'envers, mais ce n'est qu'une question de numérotation).

```
struct fraction {
  long numerateur, denominateur;
};

struct fraction reduite (long t[], int n) {
  int i;
  long p, q, tmp;
  struct fraction f;

  p = t[n];
  q = 1;
  for (i = n - 1; i >= 0; i--) {
    tmp = p;
    p = t[i] * p + q;
    q = tmp;
  }
  f.numerateur = p;
  f.denominateur = q;
  return f;
}
```

Remarque : l'initialisation p=1, q=0 est un peu moins claire, mais plus élégante :

```
long p = 1, q = 0, tmp;

for (i = n; i >= 0; i--)
  etc.
```

Exemple de calcul, qui illustre que la qualité des approximations fournies par les réduites croît très rapidement :

```
Entrer un réel et un entier: 3.141592654 5

Développement d'ordre 5 : [3 7 15 1 293]

Réduite d'ordre 0:      3 / 1     = 3.0
Réduite d'ordre 1:     22 / 7     = 3.142857142857
Réduite d'ordre 2:    333 / 106   = 3.141509433962
Réduite d'ordre 3:    355 / 113   = 3.141592920353
Réduite d'ordre 4: 104348 / 33215 = 3.141592653921
```