

```

/*
 * continued_fractions.c
 *
 * This file provides the function
 *
 *     Boolean appears_rational(    double x0, double x1,
 *                                double confidence,
 *                                long *num, long *den);
 *
 * which checks whether a finite-precision real number x known to lie
 * in the interval (x0, x1) appears to be a rational number p/q. If it
 * does, it sets *num and *den to p and q, respectively, and returns TRUE.
 * Otherwise it sets *num and *den to 0 and returns FALSE. The confidence
 * parameter gives the maximal acceptable probability of a "false positive".
 * (The following description of the algorithm explains the confidence
 * parameter more precisely.) By the way, it's OK to pass x0 and x1 in
 * the "wrong order": if x0 > x1, appears_rational() swaps them before
 * proceeding. But appears_rational() always returns FALSE for x0 == x1.
 *
 * First recall that a continued fraction expansion is an expression like
 *
 *      1
 * 48/11 = 4 + -----
 *              1
 *            2 + -----
 *                  1
 *                1 + ---
 *                   3
 *
 * The algorithm to compute a continued fraction expansion, given a
 * floating point approximation x, is roughly the following:
 *
 *     do
 *     {
 *         subtract off integer part
 *         invert fractional part
 *     } until (fractional part is zero)
 *
 * The only tricky part is the numerical analysis. Rather than working
 * with a single number x, we'll work with an interval [x0, x1] in which
 * x is known to lie (the width of the interval corresponds to the
 * precision with which we know x). The above algorithm becomes
 *
 *     while (the interval [x0, x1] does not contain an integer)
 *     {
 *         subtract off the integer part
 *         invert, i.e. map [x0, x1] -> [1/x1, 1/x0]
 *     }
 *
 * Proposition. For any interval of positive width, the algorithm
 * converges in a finite number of steps.
 *
 * Proof. Each iteration expands the width of the interval, because the
 * derivate of the function f(x) = 1/x is f'(x) = -1/x^2, which implies
 * |f'(x)| > 1 on the interval (0,1). However, what we'd really like to
 * show is that it expands the interval by some factor r > 1. Even though
 * this isn't quite true (as x -> 1, |f'(x)| -> 1), it is true that
 * two consecutive iterations must increase the width of the interval
 * by a factor of r^2, for some r > 1. To see this, note that after
 * subtracting off the integer part, the interval will lie in the range
 * (1/(n+1), 1/n] for some n (otherwise its image under the inversion
 * will include an integer and the algorithm will terminate). The first
 * inversion maps x to 1/x. 1/x lies in the range [n, n+1), so its
 * integer part is n. Subtracting off this integer part takes 1/x to
 * 1/x - n. The second inversion maps 1/x - n to 1/(1/x - n) = x/(1 - nx).
 * The derivative of this function is (1 - nx)^-2. Because x is in the
 * range (1/(n+1), 1/n], it follows that (1 - nx)^-2 > (1 - n(1/(n+1)))^-2
 * = (n+1)^2. So even in the worst case of n = 1, two iterations of the
 * mapping expand by a factor of (1+1)^2 = 4. In terms of the preceding
 * notation, we may choose r = 2. Q.E.D.
 *
 * Corollary. The number of iterations of the algorithm is bounded
 * by the number of significant binary digits in x after the decimal
 * point. Typically, of course, the number of iterations will be less.

```

```

* (However, the worst case would be realized by a floating-point
* approximation to (sqrt(5) - 1)/2.)
*
*
* We have shown that the algorithm will ALWAYS yield a rational
* approximation to x, i.e. a rational number in the interval [x0, x1].
* How do we know whether x is "really" a rational number or whether
* we just stumbled on a rational by chance? The width of the interval
* gives the probability of stumbling on a rational approximation by
* chance. For example, if we begin with an arbitrary interval (not
* chosen to represent a rational number) and after some number of
* iterations the interval has expanded to a width of 0.3, then there
* is a 30% probability that the interval will contain an integer.
* Our approach, therefore, is to assume that if a rational approximation
* is found while the interval is still small (e.g. 1e-4), then it must
* be because the original interval [x0, x1] represented a rational
* number. But if no rational approximation is found until the interval
* is large (e.g. 0.3), then we assume that we found the approximation
* by chance, and that the original interval [x0, x1] does not represent
* a rational. In practical terms, the calling function passes a value
* for the confidence parameter. If the width of the interval exceeds
* this confidence parameter before a rational approximation is found,
* then we assume that the original interval [x0, x1] does not represent
* a rational, and we return FALSE.
*/

#include "kernel.h"

/*
* MAX_ITERATIONS puts an upper bound on the number of iterations
* of the algorithm. It should never come into play -- it's only
* a safety net in case of disaster. Here's how I came up with the
* value 64. Assuming doubles have mantissas with 8 bytes or less,
* the smallest interval you'd reasonably expect to start with would
* have width at least 2^-64. As shown above, the width of the interval
* will, on average, at least double with each iteration. So after
* 64 iterations the interval would have width 1 and you'd have found
* a rational approximation (or, more likely, you'd have exceeded
* the confidence parameter).
*/
#define MAX_ITERATIONS 64

Boolean appears_rational(
    double x0,
    double x1,
    double confidence,
    long *num,
    long *den)
{
    int i,
        j;
    long a[MAX_ITERATIONS],
        p,
        q,
        t;
    double n,
        temp;

    /*
     * Make sure x0 <= x1.
     */
    if (x0 > x1)
    {
        temp = x0;
        x0 = x1;
        x1 = temp;
    }

    /*
     * Make sure the width of the interval [x0, x1] is less than
     * the confidence parameter.
     */
    if (x1 - x0 >= confidence)
    {

```

```

    *num = 0;
    *den = 0;
    return FALSE;
}

/*
 * We expect the function to return() from within this loop.
 * We should never reach i == MAX_ITERATIONS.
 */

for (i = 0; i < MAX_ITERATIONS; i++)
{
    /*
     * If x0 and x1 were initially equal, or were so close that they
     * became equal during the course of the computation, return FALSE.
     * (E.g. if x0 = -1e-15 and x1 = -1e-15 + 1e-25, then
     * x0 + 1.0 == x1 + 1.0 == 1.0 - 1e-15.)
     */
    if (x1 == x0)
    {
        *num = 0;
        *den = 0;
        return FALSE;
    }

    /*
     * Let a[i] be the integer part of x1.
     * Subtract it from both x0 and x1.
     *
     * Technical note: n is a double; a[i] is a long int.
     */

    n = floor(x1);
    a[i] = (long int) n;
    x0 -= n;
    x1 -= n;

    /*
     * x1 is now in the range [0,1).
     * [x0, x1] will contain the origin iff x0 is nonpositive.
     */

    if (x0 <= 0.0)
    {
        /*
         * Success!
         *
         * The interval [x0, x1] contains the origin,
         * yet its width is still less than the confidence parameter.
         *
         * We may now reconstruct the rational approximation from
         * the a[i].
         */

        /*
         * Set p/q = a[i], the most recently subtracted integer.
         */
        p = a[i];
        q = 1;

        /*
         * For each preceding a[j], invert p/q and add a[j].
         */
        for (j = i; --j >= 0; )
        {
            /*
             * Invert p/q.
             */
            t = p;
            p = q;
            q = t;

            /*
             * Add a[j].

```

```

        *
        *      Note that  $\frac{p}{q} + a[j] = \frac{p}{q} + \frac{q * a[j]}{q} = \frac{p + q * a[j]}{q}$ 
        */
        p += q * a[j];
    }

    /*
    * Set *num and *den and return TRUE.
    */
    *num = p;
    *den = q;
    return TRUE;
}

/*
* Map [x0, x1] to [1/x1, 1/x0].
*/
temp = x0;
x0 = 1.0 / x1;
x1 = 1.0 / temp;

/*
* If the width of the interval exceeds the confidence parameter,
* then we've failed to find a meaningful rational approximation.
*/
if (x1 - x0 > confidence)
{
    *num = 0;
    *den = 0;
    return FALSE;
}

/*
* As shown in the documentation above, an interval of
* nontrivial width must expand to width at least one
* within MAX_ITERATIONS of the algorithm. So we should
* never reach this point.
*/
uFatalError("appears_rational", "continued_fractions");

/*
* We'll never get past the uFatalError() call,
* but we must provide a return value to keep the
* compiler happy.
*/
return FALSE;
}

```