Recurrence relation evaluation

Synopsis

```
#include <boost/math/tools/recurrence.hpp>

template <class T, class NextCoefs>
inline T solve_recurrence_relation_forward(NextCoefs& get_coefs, unsigned
last_index, T& first, T& second);

template <class T, class NextCoefs>
inline T solve_recurrence_relation_backward(NextCoefs& get_coefs, unsigned
last_index, T& first, T& second);

template <class Coefficients, class U, class T>
inline T solve_recurrence_relation_by_olver(Coefficients& coefs, const U&
tolerance, unsigned index, const T& init_value);
```

Description

Many special functions satisfy second-order recurrence relations, or difference equations, of the form:

$$a_n w_{n+1} - b_n w_n + c_n w_{n-1} = d_n$$

If d(n) = 0 for any n, this equation is homogeneous, otherwise it is inhomogeneous. Equation of this form can be computed recursively for n = 2, 3, …. However for floating-point number each step of the calculation introduces rounding errors. These errors have the effect of perturbing solution by unwanted small multiples of w(n) and of an independent solution g(n). This is of little consequence if the wanted solution is growing in magnitude at least as fast as any other solution of the difference equation, and the recursion process is stable.

Two first functions *solve_recurrence_relation_forward* and *solve_recurrence_relation_backward* are intended for stable recurrence relations of homogeneous equations. Parameters of the function:

- *get_coefs* is a tuple, for which get<0>() is a(n), get<1>() is b(n), get<2>() is c(n);
- *last_index* is an index N to be found;
- *first* is w(0) value;
- *second* is w(1) value.

If w(n) is a nontrivial solution such that

$$\lim_{n\to\infty} \frac{w_n}{g_n} = 0$$

then w(n) is said to be a minimal solution and g(n) is a dominant solution. In this situation the unwanted multiples of g(n) grow more rapidly than the wanted solution, and the computations are unstable.

For this case Miller's algorithm is useful. However, there is an important improved Olver's algorithm, which has at least the following advantages:

- it's applicable to inhomogeneous difference equations as well;
- we don't need to guess starting number N for initial value to yield backward recursion for Miller's algorithm. This value is computed automatically.

For more information, see http://dlmf.nist.gov/3.6#v

Function *solve_recurrence_relation_by_olver* solves difference equation at unstable direction using Olver's algorithm. Parameters of the function:

- *coefs* is a tuple, for which get<0> is a(n), get<1>() is b(n), get<2>() is c(n) (and get<3>() is d(n) for inhomogeneous case);
- *tolerance* is the precision required;
- *index* is an index N to be found;
- *init_value* is w(0) value.

For Both Miller's and Olver's algorithms there is one subtle point that for quite large N they may suffer from possible overflows. In this case our implementation has some logarithmic conditions where recurrence stops and then repeats again for new initial value.

Example

Let's implement recurrence relation for Bessel function by known formula

$$J_{v+1}(x) = \frac{2v}{x} J_v - J_{v-1}(x)$$

For this relation backward direction is stable unlike forward. First, we should create functor type for coefficients:

```
template <class T>
struct bessel_j_coefficients_t
{
  typedef boost::math::tuple<T, T, T> result_type;

  bessel_j_coefficients_t(const T& v, const T& x):
    v(v), x(x)
  {
  }

  result_type operator()(int i) const
  {
    const T vi = v + i;

    const T an = 1;
    const T bn = (2 * vi) / x;
    const T cn = 1;

    return boost::math::make_tuple(an, bn, cn);
  }

private:
  const T v, x;
};
```

Then the usage is fairly easy:

```
int main()
{
  using namespace boost::math::tools;

  const double v = 0.1, x = 20;
  const double j0 = boost::math::cyl_bessel_j(v, x);
  const double jm1 = boost::math::cyl_bessel_j(v - 1, x);
  const bessel_j_coefficients_t<double> bessel_j_coefficients(v - 1, x);

  std::cout.precision(std::numeric_limits<double>::digits10);

  std::cout << boost::math::tools::solve_recurrence_relation_backward(
      bessel_j_coefficients,
      100u,
      j0,
      jm1); // solves besselj(-99.9, 2)

  std::cout << std::endl;

  std::cout << boost::math::tools::solve_recurrence_relation_by_olver(
      bessel_j_coefficients,
      boost::math::tools::epsilon<double>(),
      100u,
      jm1); // solves besselj(99.1, 2)
}
```