1 Cubic rootfinder using Halley's method: C/C++ program code

```
/*****************************************************************************/
/* eqn_cubic: real roots of a cubic equation                                 */
/*===========================================================================*/
/* a            (i)  array containing the polynomial coefficients            */
/* x            (o)  result array                                            */
/* RESULT       (o)  number of real roots                                    */
/*****************************************************************************/
#include <cmath>

using namespace std;

inline double sq(const double& x) {     // Remark #1
  return x * x;
}

inline int signum(const double& x) {
  return (0.0 < x) - (x < 0.0);
}

inline double newton1(const double *a, double x) {
  double y, y1;
  y = a[2] + x;
  y1 = 2 * y + x;
  y1 = x * y1 + a[1];
  y = (x * y + a[1]) * x + a[0];
  if (y1 != 0.0) x -= y / y1;
  return x;
}

inline int eqn_quadratic(const double *a, double *x, int i, int j) {
  double p = -0.5 * a[1],
    d = sq(p) - a[0];
  if (d >= 0.0) {
    d = sqrt(d);
    x[i] = p - d;
    x[j] = p + d;
    return 2;
  }
  return 0;
}
inline int eqn_quadratic(const double *a, const double *o, double *x,
int i, int j) {
  double p = -0.5 * a[1],
    d = sq(p) - a[0];
  if (d >= 0.0) {
    d = sqrt(d);
    if (p < 0.0) {
      x[i] = newton1(o, p - d);
      x[j] = p + d;
    }
    else {
      x[i] = p - d;
      x[j] = newton1(o, p + d);
    }
```

```
    return 2;
  }
  return 0;
}

int eqn_cubic(const double *a, double *x) { // Remark #2
  int i_slope, i_loc;
  double w, xh, y, y1, y2, dx, b[3], c[2], p, d, *pb;
  const double *pa,
    prec = 1.0e-6;                      // termination criterion, Remark #3

  if (a[3] == 0.0) {                    // a less-than-cubic problem?
    if (a[2] == 0.0) {
      if (a[1] == 0.0)                  // ill-defined problem
        return 0;
      else {                            // linear problem
        x[0] = -a[0] / a[1];
        return 1;
      }
    }
    else {                              // quadratic problem
      w = 1.0 / a[2];
      c[0] = a[0] * w;
      c[1] = a[1] * w;
      return eqn_quadratic(c, x, 0, 1);
    }
  }

  w = 1.0 / a[3];                       // normalize
  pa = a;
  pb = b;
  while (pb <= b + 2) *pb++ = *pa++ * w;

  if (b[0] == 0.0) {                    // root at zero? Remark #4
    x[0] = 0.0;
    if (eqn_quadratic(b + 1, x, 1, 2) == 0)
      return 1;
    else {                              // sort results
      if (x[2] < 0.0) {
        x[0] = x[1];
        x[1] = x[2];
        x[2] = 0.0;
      }
      else if (x[1] < 0.0) {
        x[0] = x[1];
        x[1] = 0.0;
      }
      return 3;
    }
  }

  xh = -1.0 / 3.0 * b[2];               // inflection point, Remark #5
  y = b[0] + xh * (b[1] + xh * (b[2] + xh));
  if (y == 0.0) {                       // is inflection point a root?
    x[0] = x[1] = xh;
```

```
    c[1] = xh + b[2];                    // deflation
    c[0] = c[1] * xh + b[1];
    return 1 + eqn_quadratic(c, x, 0, 2);
  }
  i_loc = (y >= 0.0);
  d = sq(b[2]) - 3 * b[1];
  if ((i_slope = signum(d)) == 1)        // Laguerre-Nair-Samuelson bounds
    xh += ((i_loc) ? -2.0 / 3.0 : 2.0 / 3.0) * sqrt(d);
  else if (i_slope == 0) {               // saddle point?
    x[0] = xh - cbrt(y);
    return 1;
  }

  do {                                    // iteration (Halley's method)
    y = b[2] + xh;
    y1 = 2 * y + xh;
    y2 = y1 + 3 * xh;
    y1 = xh * y1 + b[1];
    y = (xh * y + b[1]) * xh + b[0];
    dx = y * y1 / (sq(y1) - 0.5 * y * y2);
    xh -= dx;
  } while (fabs(dx) > prec * fabs(xh)); // Remark #6
  x[0] = x[2] = xh;

  if (i_slope == 1) {
    c[1] = xh + b[2];                    // deflation
    c[0] = c[1] * xh + b[1];
    return 1 + eqn_quadratic(c, b, x, i_loc, i_loc + 1);
  }
  return 1;
}
```

Remarks:

#1 `sq()` (= square), `signum()` (= sign function), `newton1()` (= single Newton iteration step), and `eqn_quadratic()` should be implemented as inline functions (the former two preferably as template functions) or preprocessor macros in order to avoid the usual overhead of a subroutine call. We need two versions of `eqn_quadratic()`, one without and one with the capability of doing a postiteration.

#2 Users of C++ might prefer to define `a` and `x` as instances of an array or vector class.

#3 Halley's method usually has a convergence order of 3, which practically means that the number of correct places in the result triples with each iteration step. If two subsequent iteration steps differ by less than 10^{-6} (relatively), the termination error of the last step is less than 10^{-18}.

#4 This code block is not strictly necessary. But if it is omitted, the convergence criterion (see Remark #6) may become inefficient for a root happening to lie at $x = 0.0$.

#5 Modern optimizing compilers evaluate constant numerical expressions, so that these floating-point divisions will not be performed at run time.

#6 Some CPU time can be saved by working with a fixed threshold. But this may not be safe under all circumstances.

2 C/C++ code snippet: normalization and scaling

input: array `a[4]`, coefficients of the original cubic polynomial
results: array `b[4]`, coefficients of the scaled polynomial ($b_3 = 1; |b_0|, |b_1|, |b_2| \leq 1$), scaling factor `sc`

```
int i;
double b[4], w, sc, *pb;
const double *pa;

w = 1.0 / a[3];                         // normalization factor
i = 1;
pa = a + 2;
pb = b + 2;
while (pb >= b) {
  *pb = *pa-- * h;                      // normalize
  f = frexp(*pb--, &e);                 // extract binary exponent
  e /= i++;
  if (e > emax) emax = e;               // find largest exponent
}
e = -emax - 1;
sc = ldexp(1.0, -e);                    // scaling factor
i = 0;
pb = b + 3;
*pb = 1.0;
while (pb >= b) {
  i += e;
  *pb = ldexp(*pb, i);                  // scale the coefficients
  pb--;
}
```