C++ algorithms from “Analytical Ballistic Trajectories with Approximately Linear Drag”, Giliam J. P. de Carpentier.

ALGORITHM 1

```cpp
#include <algorithm> //                              Included for std::min.
#include <cmath> //                                  Included for std::sqrt.

double GetTimeToTargetRWithMinimalInitialSpeed(double k, double vInfinity,
                                               double rX, double rY) {
    //   1. Start by getting coefficients for the function f(t) = a4*t^4 + a3*t^3
    //    + a1*t + a0 which is 0 at the sought time-to-target t. Solving f(t) = 0
    //   for t > 0 is equivalent to solving e(u) = f(1/u)*u^4 = a0*u^4 + a1*u^3 +
    //  a3*u + a4 = 0 for u where u = 1 / t, but the latter is more well-behaved,
    //   being a strictly concave function for u > 0 for any set of valid inputs,
    //    so solve e(u)=0 for u instead by converging from an upper bound towards
    double kVInfinity = k * vInfinity, rr = rX * rX + rY * rY;  //   the root and
    double a0 = -rr, a1 = a0 * k, a3 = k * kVInfinity * rY; //         return 1/u.
    double a4 = kVInfinity * kVInfinity;
    double maxInvRelError = 1.0E6; //      Use an achievable inverse error bound.
    double de, e, uDelta = 0;

    //    2. Set u to an upper bound by solving e(u) with a3 = a1 = 0, clamped by
    //             the result of a Newton method's iteration at u = 0 if positive.
    double u = std::sqrt(kVInfinity / std::sqrt(rr));
    if (rY < 0) u = std::min(u, -vInfinity / rY);

    //   3. Let u monotonically converge to e(u)'s positive root using a modified
    // Newton's method that speeds up convergence for double roots, but is likely
    //              to overshoot eventually. Here, 'e' = e(u) and 'de' = de(u)/du.
    for (int it = 0; it < 10; ++it, uDelta = e / de, u -= 1.9 * uDelta) {
        de = a0 * u; e = de + a1; de = de + e; e = e * u;
        de = de * u + e; e = e * u + a3; de = de * u + e; e = e * u + a4;
        if (!(e < 0 && de < 0)) break; //                       Overshot the root.
    }
    u += 0.9 * uDelta; //    Trace back to the unmodified Newton method's output.

    //   4. Continue to converge monotonically from the overestimated u to e(u)'s
    //                                  only positive root using Newton's method.
    for (int it = 0; uDelta * maxInvRelError > u && it < 10; ++it) {
        de = a0 * u; e = de + a1; de = de + e; e = e * u;
        de = de * u + e; e = e * u + a3; de = de * u + e; e = e * u + a4;
        uDelta = e / de; u -= uDelta;
    }

    //   5. Return the solved time t to hit [rX, rY], or 0 if no solution exists.
    return u > 0 ? 1 / u : 0;
}
```

ALGORITHM 2

```cpp
#include <algorithm> //                              Included for std::min.
#include <cmath> //                                  Included for std::sqrt.

double GetTimeToTargetRGivenInitialSpeedS(double k, double vInfinity, double rX,
                                          double rY, double s, bool highArc) {
    //   1. Start by getting coefficients for the function f(t) = a4*t^4 + a3*t^3
    //    + a2*t^2 + a1*t + a0 which is 0 at the sought time-to-target t. Solving
    //   f(t) = 0 for t > 0 is equivalent to solving e(u) = f(1/u)*u^3 = a0*u^3 +
    //    a1*u^2 + a2*u + a3 + a4/u for u where u = 1 / t, but the latter is more
    //    well-behaved, being a strictly convex function for u > 0 for any set of
    //  inputs iff a solution exists, so solve for e(u) = 0 instead by converging
    //          from a high or low bound towards the closest root and return 1/u.
    double kRX = k * rX, kRY = k * rY, kRXSq = kRX * kRX, sS = s * s;
    double twoKVInfinityRY = vInfinity * (kRY + kRY), kVInfinity = k * vInfinity;
    double a0 = rX * rX + rY * rY, a1 = (k + k) * a0;
    double a2 = kRXSq + kRY * kRY + twoKVInfinityRY - sS;
    double a3 = twoKVInfinityRY * k, a4 = kVInfinity * kVInfinity;
    double maxInvRelError = 1.0E6; //      Use an achievable inverse error bound.
    double maxV0YSq = sS - kRXSq;//maxV0YSq is the max squared 'V0.y' that leaves
    double e, de, u, uDelta = 0;  //      enough 'V0.x' to reach rX horizontally.

    //        2. Set u to a lower/upper bound for the high/low arc, respectively.
    if (highArc) { //  Get smallest u vertically moving rY at max possible +v0.y.
        double minusB = std::sqrt(maxV0YSq) - kRY;
        double determ = minusB * minusB - (twoKVInfinityRY + twoKVInfinityRY);
        u = (kVInfinity + kVInfinity) / (minusB + std::sqrt(determ));
        maxInvRelError = -maxInvRelError; //    Convergence over negative slopes.
    } else if (rY < 0) { // Get largest u vertically moving rY at most neg. v0.y.
        double minusB = -std::sqrt(maxV0YSq) - kRY;
        double determ = minusB * minusB - (twoKVInfinityRY + twoKVInfinityRY);
        u = (minusB - std::sqrt(determ)) / (rY + rY);
        //   Clamp the above bound by the largest u that reaches rX horizontally.
        u = std::min(s / rX - k, u);
    } else u = s / std::sqrt(a0) - k; //           Get the (largest) u hitting rX
    //         horizontally a.s.a.p. while launching in the direction of [rX,rY].

    //    3. Let u monotonically converge to e(u)'s closest root using a modified
    //   Newton's method, almost scaling the delta as if the solution is a double
    int it = 0; //    root. Note that 'e' = e(u) * u^2 and 'de' = de(u)/du * u^2.
    for (; it < 12; ++it, uDelta = e / de, u -= 1.9 * uDelta) {
        de = a0 * u; e = de + a1; de = de + e; e = e * u + a2; de = de * u + e;
        e = e * u + a3; e = (e * u + a4) * u; de = de * u * u - a4;
        if (!(u > 0 && de * maxInvRelError > 0 && e > 0)) break; //     Overshot.
    }
    u += 0.9 * uDelta; //        Trace back to unmodified Newton method's output.

    //         4. Continue to converge monotonically to e(u)'s closest root using
    //    Newton's method from the last known conservative estimate on the convex
    //      function. (Note that in practice, u will have converged enough in <12
    for (; u > 0 && it < 12; ++it) {   // iterations iff a solution does exists.)
        de = a0 * u; e = de + a1; de = de + e; e = e * u + a2; de = de * u + e;
        e = e * u + a3; e = (e * u + a4) * u; de = de * u * u - a4;
        uDelta = e / de; u -= uDelta;
        if (!(de * maxInvRelError > 0)) break; // Wrong side of the convex 'dip'.
        if (uDelta * maxInvRelError < u && u > 0) return 1 / u; //  5a. Found it!
    }

    //      5b. If no solution was found, return 0. This only happens if s (minus
    //     a small epsilon) is too small to have a solution, the target is at the
    return 0; //   origin, or the parameters are so extreme they cause overflows.
}
```