Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2014, Article ID 463489, 13 pages
http://dx.doi.org/10.1155/2014/463489

Research Article

Analytical Ballistic Trajectories with Approximately Linear Drag

Giliam J. P. de Carpentier

Linnaeusstraat 32 bis, 3553 CE Utrecht, The Netherlands

Correspondence should be addressed to Giliam J. P. de Carpentier; giliam@decarpentier.nl

Received 29 April 2013; Accepted 18 September 2013; Published 30 January 2014

Academic Editor: Jue Wang

This paper introduces a practical analytical approximation of projectile trajectories in 2D and 3D roughly based on a linear drag model and explores a variety of different planning algorithms for these trajectories. Although the trajectories are only approximate, they still capture many of the characteristics of a real projectile in free fall under the influence of an invariant wind, gravitational pull, and terminal velocity, while the required math for these trajectories and planners is still simple enough to efficiently run on almost all modern hardware devices. Together, these properties make the proposed approach particularly useful for real-time applications where accuracy and performance need to be carefully balanced, such as in computer games.

1. Introduction

A ballistic trajectory is the path of an object that is dropped, thrown, served, launched, or shot but has no active propulsion during its actual flight. Consequently, the trajectory is fully determined by a given initial velocity and the effects of gravity and air resistance. Mortars, bullets, particles, and jumping computer game characters (between key presses) are all examples of ballistics, while actively controlled aircraft and rocket-propelled grenades are not.

Describing the exact motion of an object in free fall is a classic problem that can become quite complex when including effects like drag, turbulence, height-dependent medium pressure, position-dependent gravity, buoyancy, lift, and rotation. For this research paper, the problem will be approached on a relatively pragmatic level, as it will be based on a reasonably simple drag model that does not consider the dynamics of projectile rotation and assumes that wind, gravity, and terminal velocity all remain fixed over the whole trajectory. As a result, some accuracy will be sacrificed for the sake of computational efficiency and flexibility in practical use, while still maintaining much of the essence of the ballistic motion through a resistive medium. Although such a choice does not make much sense for most scientific and military applications, it does make sense for computer games, where performance is typically more important than physical correctness.

Currently, computer games hardly ever use trajectories influenced by air resistance when complex planning is required. That might partially be because implementing a computer player that is capable of quickly calculating the ideal angle to fire a mortar with a fixed initial speed to hit a given target, for example, is harder when having to take into account drag and wind conditions. In fact, the added complexity and computational intensity that would be required for working with many of the current drag models might simply not be justifiable.

This paper introduces a trajectory model that is designed to fit in the gap where working with accurate models would be too complex and working with simple dragless parabola-shaped trajectories would be insufficient. The proposed model's use and practicality will be demonstrated by covering a number of its properties and showing how these trajectories can be planned in a variety of ways.

In Section 2, previous work is shortly considered. In Section 3, the new model will be introduced and qualitatively compared to other models. In Section 4, the first planner is covered. Other planners use a different space as explained in Section 5 and will be covered in Section 6. This is followed by a discussion of Future Work in Section 7, the Conclusion in

```
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 1: Specialized C++ Quartic Solver.

Section 8, and the References. The included Algorithms 1 and 2 contain C++ functions that efficiently solve the two most complex planning problems.

2. Previous Work

The motion of ballistic projectiles has been covered in many physics papers and textbooks, and all of these use their own set of assumptions to create an approximate model of the forces acting on a projectile. Although some ballistics research focuses on effects like shape and orientation [1], spin [2], or (sub)orbital flight [3], most works on ballistic trajectories assume a fixed gravitational pull and use a simplified drag model. These drag models typically ignore all effects of in-flight rotation and are only dependent on the local velocity relative to the medium and on a given fixed terminal velocity or drag coefficient.

The used drag model influences both realism and computational complexity. For example, when no drag force is applied, the trajectory will always be a parabola, which is easy to work with and plan for. If the drag force is chosen to be linear in velocity, an explicit function describing the trajectory can be found by solving a set of linear differential equations [4]. This transcendental function is already computationally harder to calculate and even harder to plan with (i.e., solve for) [5]. To approximate reality even better, the drag force can be made quadratic in the object's velocity relative to the medium. But as no exact analytic solution for the resulting trajectory exists, calculating a trajectory requires either crude approximation or numerical integration, and planning a trajectory requires reiteration [6–8].

The research in this paper is based on a novel approximation of the trajectory function that follows from the linear drag model, sacrificing some of its moderate accuracy for a further increase in both efficiency and flexibility.

3. The Approximated Trajectory

3.1. Ballistic Parameters. Before presenting the proposed trajectory function and its properties, the necessary ballistic

```
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.
}
```

Algorithm 2: Specialized C++ Quartic Solver.

parameters will be defined here first. Starting with a general note, vector variables in this paper are always distinguished from scalar variables by the $\rightarrow$ symbol above their names. Also, the length of any vector $\vec{a}$ is denoted as $\|\vec{a}\|$, which is equal to $\sqrt{\vec{a} \cdot \vec{a}}$, where $\cdot$ is the dot product.

A ballistic object is assumed to be launched from the initial position $\vec{p}_0$ with the initial velocity $\vec{v}_0$ at time $t = 0$. Furthermore, the object will be travelling through a medium (e.g., the air) which itself travels at the fixed (wind) velocity $\vec{v}_{\text{medium}}$. It is also pulled by gravity at the fixed gravitational acceleration g, which is roughly $9.81\,\text{m/s}^2$ for "earthly" applications. The amount of drag k while moving through the medium is defined as follows.

$$k = \frac{1}{2} \frac{g}{\|\vec{v}_{\text{terminal}}\|}. \tag{1}$$

Here, $\vec{v}_{\text{terminal}}$ is the invariant terminal velocity relative to $\vec{v}_{\text{medium}}$ that is reached eventually as the forces of gravity and air resistance finally cancel each other out. On earth, that is equivalent to saying that $\vec{v}_{\text{terminal}}$ is the fixed velocity that is approached when the object is dropped from an enormous height on a windless day. Lastly, the absolute terminal velocity $\vec{v}_\infty$, being the absolute velocity approached when time t goes to ∞, is therefore

$$\vec{v}_\infty = \vec{v}_{\text{terminal}} + \vec{v}_{\text{medium}}. \tag{2}$$

Together, $\vec{p}_0$, $\vec{v}_0$, $\vec{v}_\infty$, and k uniquely define a trajectory in the proposed model. To give a real-world example of the parameters defined above, suppose a tennis ball with a terminal velocity of 30 m/s is served at 50 m/s at a 30° angle from the 381 m high roof of the Empire State building into a 10 m/s horizontal wind in 2D. Then, $\vec{p}_0 = [0, 381]$, $\vec{v}_0 = [50\cos(30^\circ), 50\sin(30^\circ)] \approx [43.30, 25]$, $k = 0.1635$, and $\vec{v}_\infty = [-10, -30]$. The trajectory resulting from these values is shown in Figure 1.

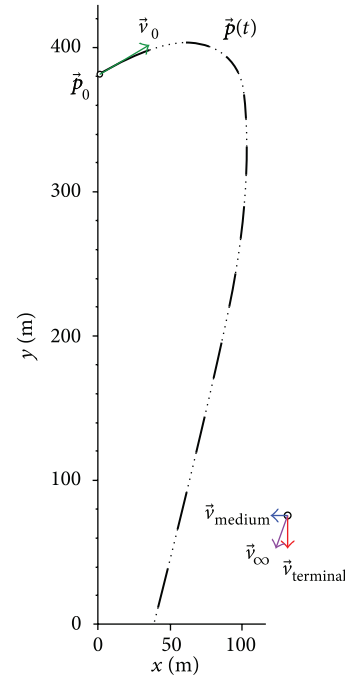

Figure 1: Serving a tennis ball from the Empire State Building upwind.

3.2. Deriving the Trajectory Function. In terms of the parameters defined above, the differential equation for the exact linear drag model has the following analytic solution:

$$\vec{p}_{\text{linear}}(t) = \frac{1}{2k}\left(\vec{v}_0 - \vec{v}_\infty\right)\left(1 - e^{-2kt}\right) + \vec{v}_\infty t + \vec{p}_0. \tag{3}$$

This function calculates the 2D or 3D position $\vec{p}_{\text{linear}}$ on a trajectory at time t where $t \geq 0$. The above function is far from new and will not be explained here in detail, as it has already been covered in many textbooks [4], occasionally even targeting game developers in particular [9] (albeit with slightly different notation and parameter definitions).

The function above will not be used directly in this paper. Instead, it will be approximated by substituting its exponential function e^x with the first degree rational function $(2 + x)/(2 - x)$ shown in Figure 2. One of the reasons for selecting this approximation over all possible other approximations to e^x is that it has a value, first derivative and second derivative that match those of e^x at $x = 0$. This means that it approximates e^x near $x = 0$ well and therefore will guarantee a good approximation of $\vec{p}_{\text{linear}}(t)$ near $t = 0$. Furthermore, the first derivative of $(2 + x)/(2 - x)$ monotonically decreases from 1 to 0 as x tends from 0 to $-\infty$, similar to the first derivative of e^x itself. When used to approximate e^x in $\vec{p}_{\text{linear}}(t)$, this property will cause the initial velocity to monotonically converge to the terminal velocity over time. Note that no polynomial approximation of e^x has this specific property, and of all possible rational functions that do possess the above properties, the proposed approximation is the simplest and thus the most efficient. Lastly, its inverse is also a first degree rational function, resulting in relatively simple algebraic solutions for all (otherwise algebraic) equations that use it to approximate e^x.

When the exponential function in $\vec{p}_{\text{linear}}(t)$ is substituted by the rational approximation, the following function is the result

$$\vec{p}(t) = \frac{\left(\vec{v}_0 + kt\vec{v}_\infty\right)t}{1 + kt} + \vec{p}_0. \tag{4}$$

Because of the aforementioned properties, $\vec{p}(t)$ will not only be more efficient to compute on modern computers, but it will also still share many of its characteristics with $\vec{p}_{\text{linear}}(t)$ and allow trajectory planning to be done with relative ease. The remainder of this paper mainly revolves around exploring these and other properties of $\vec{p}(t)$ together with their implications.

3.3. A Qualitative Comparison. As $\vec{p}(t)$ is only an approximation, it will differ from the linear drag model's trajectory function it is based on, as well as from the results of other models. For comparison purposes, the trajectories that follow from launching three different sport balls using four different

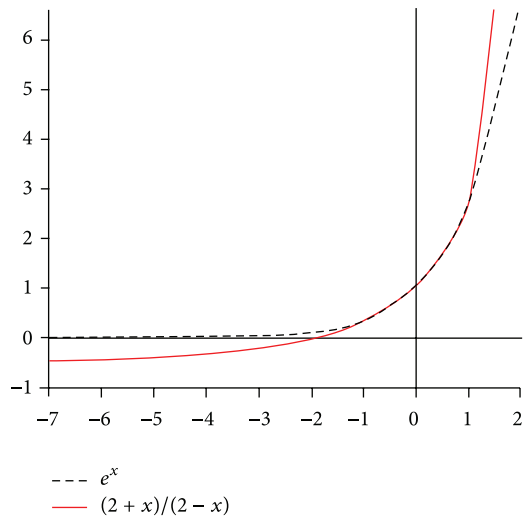

FIGURE 2: Approximating e^x using a simple rational function.

models are plotted side by side in Figure 3. All balls are launched at an 45° angle at 50 m/s on a windless earthly day ($g = 9.81\,\text{m/s}^2$). The different drag models are calibrated to respect the respective ball's terminal velocities (except for the dragless model, which always has an infinite terminal velocity).

Each of the alternating thick and thin segments in the trajectories shown in Figure 3 (and in all other trajectory plots in this paper, for that matter) represents a projectile's movement over a period of exactly one second, making it possible to not only compare the shapes of the trajectories but also their local speeds. The results from the novel $\vec{p}(t)$ function are plotted in black, the results from the linear drag model $\vec{p}_{\text{linear}}(t)$ are plotted in blue, and the results of the physically (most) correct quadratic drag model simulation $\vec{p}_{\text{quadratic}}(t)$ are plotted in green. Lastly, the red parabola $\vec{p}_{\text{dragless}}(t)$ represents the trajectory of each of the three balls in a perfect vacuum (in other words, when there is no drag and $\vec{v}_\infty$'s length goes to infinite).

When comparing the trajectories for $\vec{p}(t)$ to the results of the two more accurate drag models, they are certainly different but they still reasonably mimic these in look, feel, and properties. Consequently, the proposed model is physically at least quite plausible and is probably accurate enough for most computer game purposes. Furthermore, in some cases, $\vec{p}(t)$ is actually closer to $\vec{p}_{\text{quadratic}}(t)$ than $\vec{p}_{\text{linear}}(t)$, making it in these cases arguably even physically more accurate than the model it is approximating to. Lastly, the trajectories for all three drag models perfectly approach $\vec{p}_{\text{dragless}}(t)$ when $\vec{v}_\infty$'s length goes to infinite.

3.4. Exploring Some of p(t)'s Properties. The function $\vec{p}(t)$ given by (4) can be factored, solved, and parameterized in many different ways. For example, basic algebra allows it to be written as $\vec{p}(t) = a\vec{v}_0 + \vec{b}$ as well, where $a = t/(1 + kt)$ and $\vec{b} = \vec{v}_\infty(t - a) + \vec{p}_0$. Note that in this form, the initial velocity is separated from all other factors, and it becomes immediately clear that $\vec{p}(t)$ is a linear function in terms of $\vec{v}_0$. This implies that when launching multiple objects at some $t = 0$ with all properties equal except for the initial velocity, each of these objects has the same value for a and for $\vec{b}$. This feature may be exploited in particle explosion systems on modern GPUs, for example, requiring only one evaluation of a and $\vec{b}$ per frame per explosion (layer) on the CPU and one MAD (multiply-and-add) GPU instruction per particle per frame to calculate each particle's position.

The linearity of $\vec{p}(t)$ in terms of the initial velocity $\vec{v}_0$ can also be used for many other purposes. For example, in Figure 4, $\vec{p}(t)$ is used to calculate the green and blue positions for two different "extreme" initial velocities, which are interpreted as the top-left and bottom-right positions of a textured rectangle or *quad*. Note that the (signed) size of the quad is thus simply $\vec{p}_{\text{green}}(t) - \vec{p}_{\text{blue}}(t) = (a\vec{v}_{0,\text{green}} + \vec{b}) - (a\vec{v}_{0,\text{blue}} + \vec{b}) = a(\vec{v}_{0,\text{green}} - \vec{v}_{0,\text{blue}})$. Furthermore, all the bilinearly interpolated texels within this quad, including the red one, will move over $\vec{p}(t)$ trajectories themselves as well by virtue of the linearity in terms of the initial velocities. In other words, each texel will follow a $\vec{p}(t)$ trajectory with some initial velocity that is interpolated bilinearly between the different extreme initial velocities, as if the individual texels themselves are under direct control of a physics simulation. Consequently, it should be physically plausible to use the above to scale sprites and billboards of, for example, simple smoke (for which $\vec{v}_\infty$ would typically be upwards to simulate positive buoyancy), explosion debris, and fireworks.

Many other useful properties can easily be derived from $\vec{p}(t)$ as well. For example, the velocity $\vec{v}(t)$ of $\vec{p}(t)$ is

$$\vec{v}(t) = \frac{\partial \vec{p}(t)}{\partial t} = \frac{\vec{v}_0 + kt(2 + kt)\vec{v}_\infty}{(1 + kt)^2}. \tag{5}$$

The nonnegative time $t_{\text{top},\vec{n}}$ at which the trajectory hits its maximum in the direction of a given unit-length vector $\vec{n}$ can be found by solving $\vec{v}(t) \cdot \vec{n} = 0$ for t assuming that the direction to find the maximum in is pointing away from $\vec{v}_\infty$ (i.e., $\vec{v}_\infty \cdot \vec{n} < 0$). If that assumption is false, then the top will be at time $t = 0$. The solution to both cases is summarized by the following formula:

$$t_{\text{top},\vec{n}} = \frac{\sqrt{1 - \min\left(0, (\vec{v}_0 \cdot \vec{n}) / (\vec{v}_\infty \cdot \vec{n})\right)} - 1}{k}. \tag{6}$$

This $t_{\text{top},\vec{n}}$ may be used with (4) to get the trajectory's maximum position in the $\vec{n}$ direction. Note that when $\vec{n}$ is axis-aligned, the dot products in (6) can be optimized away. For example, when $\vec{n}$ is equal to the $+y$ axis, then $t_{\text{top},\vec{n}}$ becomes

$$t_{\text{top},y} = \frac{\sqrt{1 - \min\left(0, \vec{v}_{0,y}/\vec{v}_{\infty,y}\right)} - 1}{k}. \tag{7}$$

See Figure 5 for an example of $\vec{v}(t)$ and $\vec{p}(t_{\text{top},y})$.

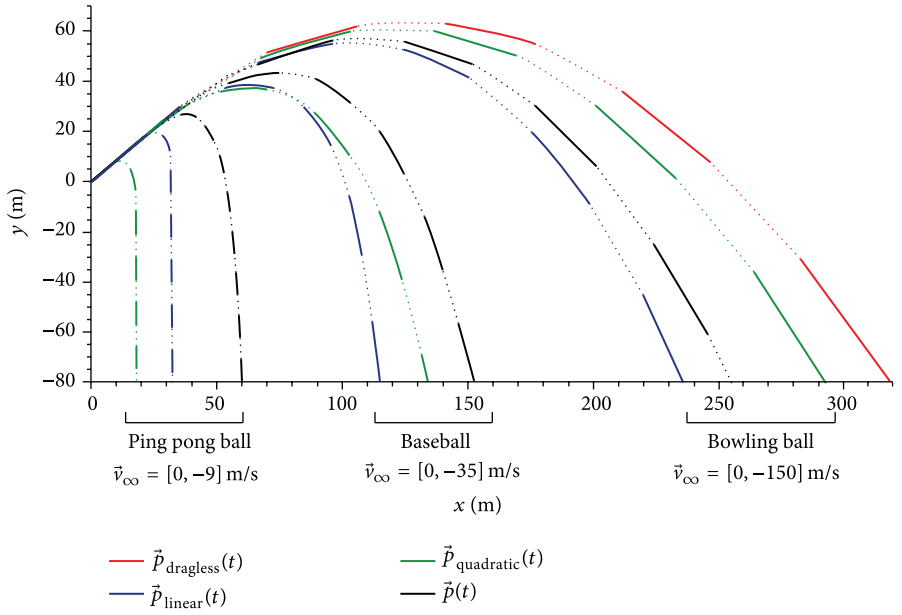

Figure 3: The trajectories resulting from four models for three different ball types.

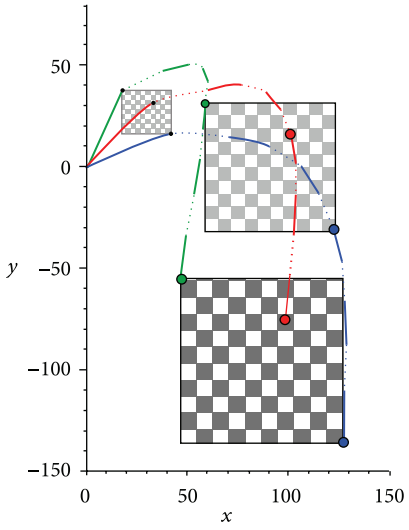

Figure 4: The trajectory of a textured quad.

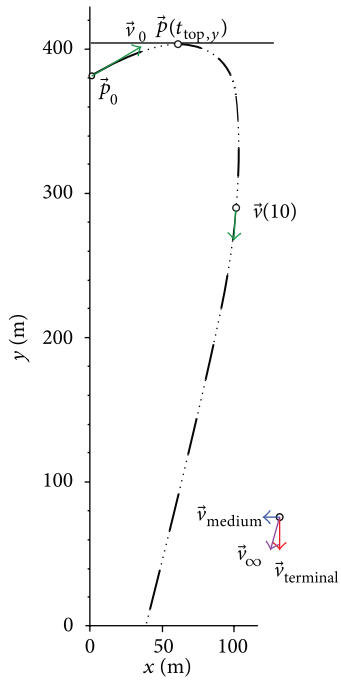

Figure 5: The velocity and y-top of a trajectory.

4. Planning in World Space: Hitting a Target at a Given Time

When a projectile needs to hit some given target position, $\vec{p}(t)$ can be used to solve or "plan" the initial velocity that leads to precisely hitting this target at some given future point in time. To be more specific, when trying to hit some position $\vec{r}$ at the given time t_r, the solution is found by solving $\vec{p}(t_r) = \vec{r}$ for $\vec{v}_0$, which results in the following formula:

$$\vec{v}_0 = k\left(\vec{r} - \vec{p}_0 - \vec{v}_\infty t_r\right) + \frac{\vec{r} - \vec{p}_0}{t_r}. \quad (8)$$

In Figure 6, this formula is used to plan the trajectories to six different target $\vec{r}$ positions, all taking exactly ten seconds to reach their target. In other words, $t_r = 10$, which results in having exactly five thick and five thin segments per trajectory in this figure.

One interesting property of this function is that the x and y components (or the x, y, and z components in the 3D case) of $\vec{v}_0$ are completely independent from each other. As a direct consequence, similar projectiles that target the

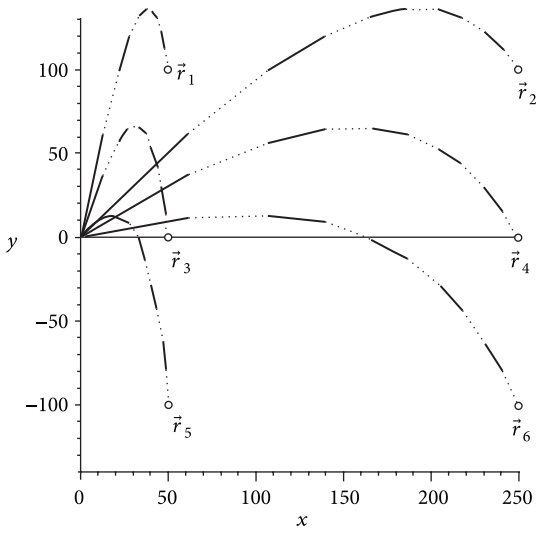

Figure 6: Planning to hit all six different target positions in exactly ten seconds.

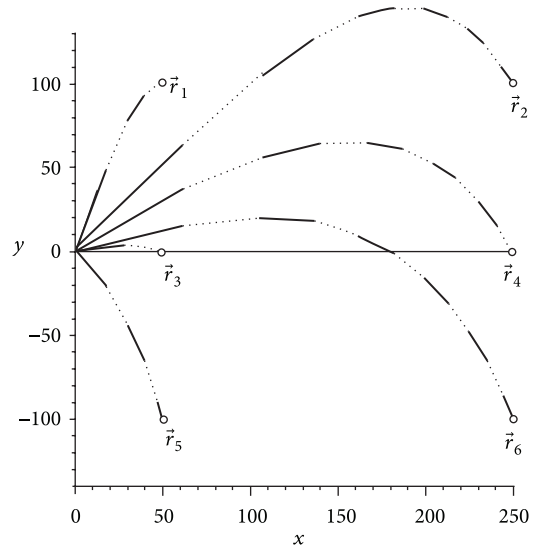

Figure 7: Planning trajectories to six targets, taking the exact same time as travelling with 25 m/s over the straight line from initial to target position.

same horizontal distance but a different height will always move at the same horizontal speed over the whole trajectory and vice versa. Also, trajectories for targets at equal height but different horizontal distances will all have the same top height, as can be observed in Figure 6 as well.

Note that the given time parameter t_r could be any positive value, including the one that is dependent on another function of $\vec{r}$. One simple example of such a function is $t_r = \|\vec{r} - \vec{p}_0\|/v_{\text{straight}}$, where v_{straight} is a given average speed over the straight line from $\vec{p}_0$ to $\vec{r}$. In Figure 7, this is illustrated using $v_{\text{straight}} = 25$ m/s.

A function for t_r could be made arbitrarily complex. The time planners that will be explored in Section 6 are examples of moderately complex functions, solving different additional constraints for t_r given $\vec{r}$. But even more complex planners would be necessary if t_r and $\vec{r}$ were to be dependent on each other. This can happen, for example, when a moving target needs to be hit while planning to launch at a fixed speed, making t_r influence the prediction of the future target position $\vec{r}$, which influences t_r again. These relationships are not explored in detail in this paper, but it is worth mentioning that some of these problems may be solved iteratively by starting with a rough estimate for t_r and then letting it converge to the right solution by repeatedly going from t_r to $\vec{r}$ and from $\vec{r}$ to an improved t_r. These iterations could possibly even be spread over multiple frames to amortize costs, for example, improving accuracy with each new frame.

5. The Principal Frame of Reference and Its Properties

Most planners for $\vec{p}(t)$ are still to be presented. However, as these planners depend on a special frame of reference to keep the required planner math as simple as possible, this frame of reference will be covered here first.

Inspecting $\vec{p}(t)$ reveals that the function always outputs $\vec{p}_0$ plus a linear combination (i.e., a weighted sum) of $\vec{v}_0$ and $\vec{v}_\infty$. Geometrically, this implies that all trajectories, even with wind coming from any 3D direction, are guaranteed to lie on a plane spanned by $\vec{v}_0$ and $\vec{v}_\infty$ which passes through $\vec{p}_0$. Consequently, $\vec{p}(t)$ may be rewritten as

$$\vec{p}\,(t) = x\,(t)\,\vec{X} + y\,(t)\,\vec{Y} + \vec{p}_0, \tag{9}$$

where $\vec{X}$ and $\vec{Y}$, respectively, describe an orthonormal tangent and bitangent direction in *world space* of the plane over which the projectile moves. Within this plane's 2D frame of reference, the vector $[x(t), y(t)]$ describes the movement on the trajectory over time relative to $\vec{p}_0$ and in terms of this alternative x axis (i.e., the tangent) and y axis (i.e., the bitangent).

Put into terms perhaps more familiar to computer graphics and game developers, the 2D function $[x(t), y(t)]$ is like a procedural UV coordinate that defines a projectile's location on an unwrapped plane which maps UV $[0, 0]$ to $\vec{p}_0$ and has an orthonormal tangent vector $\vec{X}$ and bitangent vector $\vec{Y}$. All this is shown in Figure 8, where the trajectory $\vec{p}(t)$ is visualized as the intersection between the described plane (on which is thus lies) and another curved surface.

Although there is an infinite amount of ways to define the $\vec{X}$ and $\vec{Y}$ vectors, the following definitions are used in this paper for their particularly useful properties:

$$\vec{X} = \frac{\vec{d} - \left(\vec{d} \cdot \vec{Y}\right)\vec{Y}}{\left\|\vec{d} - \left(\vec{d} \cdot \vec{Y}\right)\vec{Y}\right\|},$$
$$\vec{Y} = \frac{-\vec{v}_\infty}{v_\infty}. \tag{10}$$

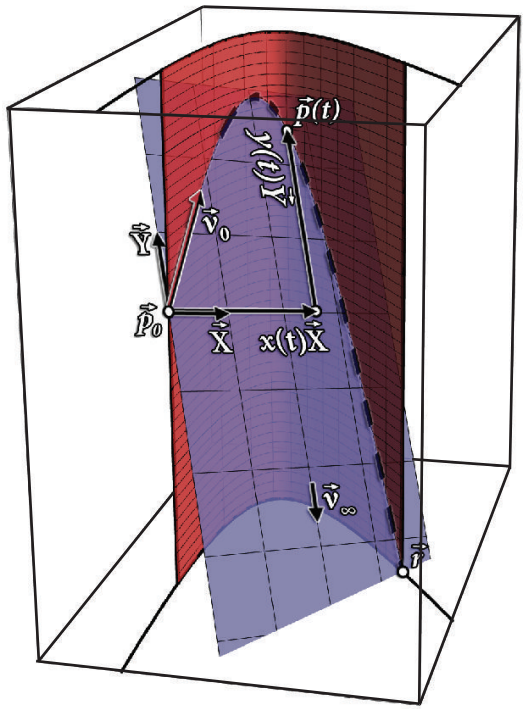

FIGURE 8: A 3D trajectory with wind at an angle, and its decomposition on the principal plane (in blue).

Here, v_∞ is used as a shorthand for $\|\vec{v}_\infty\|$. And $\vec{d} = \vec{v}_0$ if $\vec{v}_0$ is already known. In the case that $\vec{v}_0$ is not (yet) known, any position relative to $\vec{p}_0$ known to be lying on the trajectory may be used for $\vec{d}$ instead. For example, use $\vec{d} = \vec{r} - \vec{p}_0$ when targeting the position $\vec{r}$. Note that in the case that $\vec{v}_\infty$ and $\vec{d}$ are collinear, the trajectory can be described by movement solely in the $\vec{Y}$ direction. To still get a valid 2D basis in that case, an arbitrary vector that is noncollinear should be used for $\vec{d}$ instead.

The frame of reference defined by $\vec{p}_0$, $\vec{X}$, and $\vec{Y}$ is what will be called the trajectory's *principal space*. It is called that because this space allows the math describing the trajectory to be decomposed into a particularly compact and well-behaved form. In particular, wind and gravity do not affect movement over the principal $\vec{X}$ axis at all, but solely over the principal $\vec{Y}$ axis. This allows the function $x(t)$ in (9) to be simplified as will be shown and put to good use soon.

The second useful property of this principal space is that it guarantees that $v_x \geq 0$. That is, all initial and in-flight velocities expressed in principal space are guaranteed to have nonnegative values on the x axis, causing ballistic objects to never move to or be on the negative x side of this space, even though they obviously can still move in any direction in world space. Similarly, targets are never on the negative x side in principal space either. The advantage of this property is that it once again will allow for further simplifications in some of the planner math that is still to be discussed.

The third useful property of this particular space is that it is easy to convert from world space to principal space and back as $\vec{X}$ and $\vec{Y}$ are orthonormal. Converting from any world space position $\vec{p}$ to the position $[x, y]$ in principal space and vice versa can simply be done using (11) and (12), respectively as follows:

$$[x, y] = \left[(\vec{p} - \vec{p}_0) \cdot \vec{X}, (\vec{p} - \vec{p}_0) \cdot \vec{Y}\right], \tag{11}$$

$$\vec{p} = x\vec{X} + y\vec{Y} + \vec{p}_0. \tag{12}$$

Similarly, it is possible to efficiently convert from any world space velocity $\vec{v}$ to the principal velocity $[v_x, v_y]$ and back using (13) and (14), respectively as follows:

$$\left[v_x, v_y\right] = \left[\vec{v} \cdot \vec{X}, \vec{v} \cdot \vec{Y}\right], \tag{13}$$

$$\vec{v} = v_x\vec{X} + v_y\vec{Y}. \tag{14}$$

Starting a new notational convention here for clarity, vector names (i.e., variables decorated with a $\rightarrow$ symbol) are only used for variables in world space, while variables in principal space never use this decoration and always represent individual scalar quantities. So, for example, $\vec{v}_x$ is the x component of the vector representing the world space velocity $\vec{v}$, while v_x is a scalar representing a velocity in the x direction in principal space.

Now that the frame of reference itself has been covered, it is possible to define the two scalar functions that make up the principal space trajectory function $[x(t), y(t)]$:

$$x(t) = \frac{v_{0,x}t}{1 + kt}, \tag{15}$$

$$y(t) = \frac{\left(v_{0,y} - ktv_\infty\right)t}{1 + kt}. \tag{16}$$

These functions are derived by transforming $\vec{p}(t)$ into this space using (11). Note that gravity and wind do indeed not affect movement over the x direction in this space. And $[x(0), y(0)]$ is equal to $[0, 0]$, which means that trajectories in principal space always start at the origin (while starting at $\vec{p}_0$ in world space).

The simpler formula for $x(t)$ makes it possible to uniquely invert the function to get the time t at which the x component of a certain position will be reached given the horizontal initial velocity. The solution is as follows:

$$t = \frac{x}{v_{0,x} - kx}. \tag{17}$$

Here, $0 \leq x \leq v_{0,x}/k$, as that is the valid range of $x(t)$ for $t \geq 0$ as defined by (15). By plugging (17) into (16), the following explicit y-for-x relationship is found:

$$y(x) = \frac{\left(kv_\infty x/\left(kx - v_{0,x}\right) + v_{0,y}\right)x}{v_{0,x}}. \tag{18}$$

This function always has exactly one y value for each valid x value, which would not necessarily be true for an explicit trajectory function in any other space. This

property is demonstrated in Figure 9, showing a trajectory that is equivalent to the trajectory shown in Figure 1 but which is now plotted using $y(x)$ in principal space, perfectly overlaying the original trajectory when mapped back into world space.

For completeness, the principal space counterparts of the world space properties described by (5), (6), and (7) are given here as well. Consider

$$v_x(t) = \frac{v_{0,x}}{(1+kt)^2},$$
$$v_y(t) = \frac{v_{0,y} - kt(2+kt)v_\infty}{(1+kt)^2},$$
$$t_{\text{top},\vec{n}} = \frac{\sqrt{1+\max\left(0, \left[v_{0,x}, v_{0,y}\right]\cdot\left[n_x, n_y\right]/\left(v_\infty n_y\right)\right)} - 1}{k}$$
$$t_{\text{top},y} = \frac{\sqrt{1+\max\left(0, v_{0,y}/v_{\infty,y}\right)} - 1}{k}. \tag{19}$$

Lastly, the local slope in principal space in terms of time (i.e., $v_y(t)/v_x(t)$) and in terms of x (i.e., $\partial y(x)/\partial x$) is as follows:

$$\frac{v_y(t)}{v_x(t)} = \frac{v_{0,y} - kt(2+kt)v_\infty}{v_{0,x}},$$
$$\frac{\partial y(x)}{\partial x} = \frac{v_{0,y} + v_\infty}{v_{0,x}} - \frac{v_{0,x}v_\infty}{\left(kx - v_{0,x}\right)^2}. \tag{20}$$

6. Planning in Principal Space

As the planners in this section all depend on the properties of trajectories in principal space, the most relevant properties are briefly repeated here. Per definition, any ballistic object in principal space is launched from the origin, any target has a nonnegative x component, and the combined effect of gravity and wind results in a v_∞ value that is exactly in the $-y$ direction. As the planners will expect their parameters to be specified in principal space, the parameters of any world space problem need to be converted to this space before they can be used. To recap the necessary steps (assuming the problem involves hitting some target position $\vec{r}$), start by defining the actual principal space's $\vec{X}$ and $\vec{Y}$ axes using (10) with $\vec{d} = \vec{r} - \vec{p}_0$. Next, convert $\vec{r}$ (or any other requested position) to principal space using (11) to get $[r_x, r_y]$.

All planners covered here will return the exact time t_r at which the target $[r_x, r_y]$ must be hit to meet the planner's given constraints. To get the actual initial velocity in principal space that leads to hitting $[r_x, r_y]$ at this t_r, both $p_x(t_r) = r_x$ and $p_y(t_r) = r_y$ need to be solved for $[v_{0,x}, v_{0,y}]$, which can be done using the following two formulas:

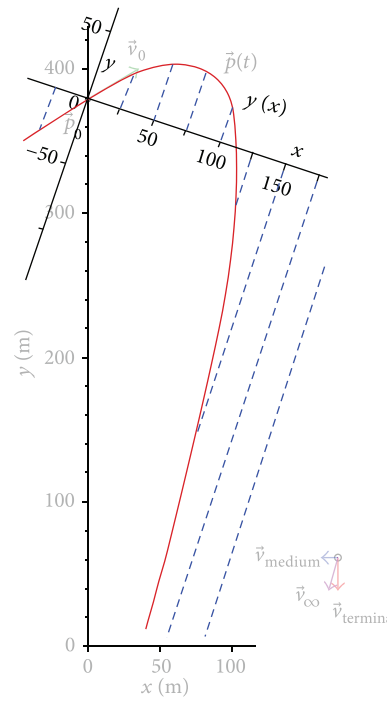

Figure 9: The same Empire State Building trajectory example as from Figure 1 but seen from the principal frame of reference.

$$v_{0,x} = r_x\left(k + \frac{1}{t_r}\right), \tag{21}$$
$$v_{0,y} = r_y\left(k + \frac{1}{t_r}\right) + kv_\infty t_r. \tag{22}$$

To get the initial velocity in world space from these, it is possible to convert $[v_{0,x}, v_{0,y}]$ to $\vec{v}_0$ using (14). But $\vec{v}_0$ may also be directly calculated from t_r and $\vec{r}$ through (8). Now that it is clear how to make use of principal space planners in general, the actual planners are presented.

6.1. Hitting the Target Given Another Position to Pass Through. A trajectory can be planned to pass through both position $[q_x, q_y]$ and through target $[r_x, r_y]$ by solving $y(q_x) = q_y$ and $y(r_x) = r_y$ for $v_{0,x}$, and using (17) on r_x and $v_{0,x}$ to get t_r. This specific form of planning may be useful to shoot through a hole or exactly over an object at $[q_x, q_y]$ to hit $[r_x, r_y]$, example. The solution to at what time the position $[r_x, r_y]$ needs to be hit is as follows:

$$t_r = s + \sqrt{s\left(s + \frac{2r_x}{k\left(r_x - q_x\right)}\right)}, \tag{23}$$

where $s = (r_x q_y - r_y q_x)/2v_\infty q_x$. Note that the line from the origin to the target position with the smallest x must be at least as steep as the line from the origin to the position with

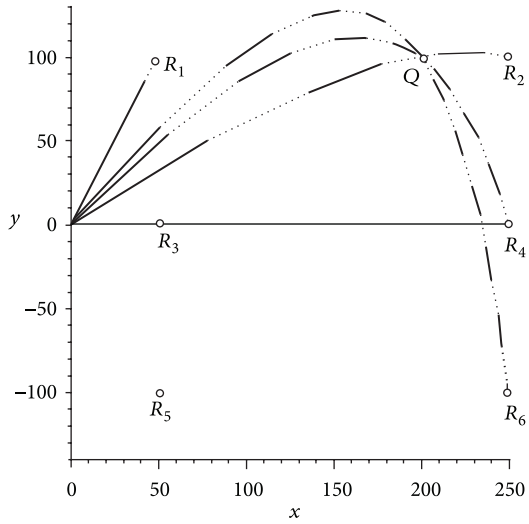

Figure 10: Planning trajectories through Q and six different R targets.

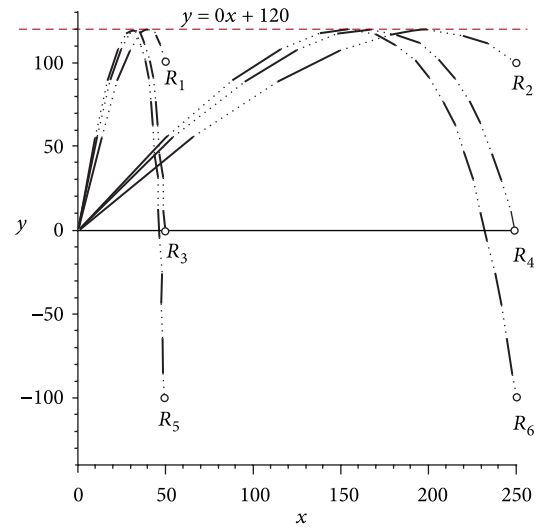

Figure 11: Planning trajectories through six different targets, touching a line at each top.

the largest x for a valid trajectory (and thus a real solution) to exist. That is, $q_y/q_x \geq r_y/r_x$ if $q_x < r_x$, and $r_y/r_x \geq q_y/q_x$ if $q_x > r_x$. In Figure 10, two of the six targets do not meet this requirement, explaining why there are only four trajectories there.

6.2. Hitting the Target While Touching a Line. When looking for the time t_r at which a trajectory passes through $[r_x, r_y]$ and touches the line $y = ax + b$ in principal space, $\partial y(x)/\partial x = a$ has to be solved for x first, which can then be used to solve $y(x) = ax + b$ for $v_{0,x}$. Both $v_{0,x}$ and r_x can then be used again with (17) to get t_r. The solution may be written as follows.

$$t_r = \frac{s + \sqrt{s^2 + kv_\infty \left(ar_x - r_y\right)}}{kv_\infty}, \tag{24}$$

where $s = \sqrt{bkv_\infty} + (1/2)k(ar_x - r_y + b)$. Note that a real solution can only exist when $b \geq 0$ and $ar_x + b \geq r_y$. That is, the line must always pass through or be above the initial and the target position.

In the example presented in Figure 11, the line to be touched is chosen to be horizontal, leading to a specification of the trajectories' vertical tops in principal space. But it is also possible to find the t_r that leads to hitting a top defined in another space. For example, to let a trajectory's top touch the world-space plane with normal $\vec{n}$ and through point $\vec{q}$, $y = ax + b$ becomes the line in principal space that describes the intersection between this plane and the trajectory's principal plane. In that case, $a = -(\vec{X} \cdot \vec{n})/(\vec{Y} \cdot \vec{n})$ and $b = ((\vec{q} - \vec{p}_0) \cdot \vec{n})/(\vec{Y} \cdot \vec{n})$. As always, when $\vec{n}$ is axis-aligned, the dot products can be optimized away. For example, when only interested in trajectories exactly hitting a world space height h at their tops (in the $+y$ direction), this simplifies to $a = -\vec{X}_y/\vec{Y}_y$ and $b = (h - \vec{p}_{0,y})/\vec{Y}_y$.

The principal space slope a can also be calculated from the slope w of a world-space elevation angle θ_{world} (i.e., $w = \tan(\theta_{\text{world}})$) by using the conversion formula $a = (\text{sign}(\vec{Y}_y)w\sqrt{s(\vec{X}_y^2 + \vec{Y}_y^2) - w^2} - s\vec{X}_y\vec{Y}_y)/(s\vec{Y}_y^2 - w^2)$, where $s = 1 + w^2$ and $\text{sign}(x) = [x \geq 0] - [x < 0]$. This a is only valid if an equivalent elevation with a positive x component in principal space exists, which is the case if $s(\vec{X}_y^2 + \vec{Y}_y^2) - w^2 \geq 0$ and $[1, w] \cdot [\sqrt{1 - \vec{X}_y^2}, \vec{X}_y] > 0$. This world-space elevation conversion may be particularly useful when used together with the next two principal-space planners.

6.3. Hitting the Target Given the Initial Slope. This subsection is about finding the time t_r at which a projectile will hit position $[r_x, r_y]$ while being launched at slope a. Planning this way may be useful when there is control over the projectile's initial speed but not over its direction (e.g., for some weapon mounted on a fixed rig).

This problem is actually simply a special case of the previous planner, where $b = 0$, meaning that the problem is equivalent to finding the trajectory that touches the line $y = ax$. After substituting b in (24) and applying some basic algebra, the solution may be written more compactly as follows:

$$t_r = \frac{s + \sqrt{s\left(s + 2v_\infty\right)}}{kv_\infty}, \tag{25}$$

where $s = (1/2)k(ar_x - r_y)$. Obviously, the slope a has to be steeper than the line from the origin to the target position (i.e., $a > r_y/r_x$) for a solution to exist, which is the case for all but one target position in Figure 12.

6.4. Hitting the Target Given the Target Slope. Similarly, it is possible to hit $[r_x, r_y]$ given the exact slope a at the target position. This type of planning allows for exact control over the angle at which a target is hit. Again, this can be seen as a

special case of the planner from Section 6.2, using $r_y - ar_x$ as the value for b. When substituted and simplified, this results in the following more direct formula:

$$t_r = \sqrt{\frac{r_y - ar_x}{kv_\infty}}. \tag{26}$$

For this particular planner, the slope a needs to be less than the slope of the line from the origin to the target position (i.e., $a < r_y/r_x$). Consequently, only five of the six targets have a valid trajectory in Figure 13.

6.5. Hitting the Target Given the Arc Height or "Curviness". The time t_r can also be calculated for a target position $[r_x, r_y]$ and an "arc height" b. Here, b is defined as the maximum difference in the y direction between the trajectory and the straight line from origin to target. Equivalently, b may be interpreted as the height in principal space of the smallest parallelogram containing the whole trajectory, as shown in Figure 14. This problem is another special case of the "line touching" problem and can be solved by substituting r_y/r_x for a in (24). After applying some basic algebra, the resulting formula may be written as follows:

$$t_r = \frac{b}{v_\infty} + 2\sqrt{\frac{b}{kv_\infty}}. \tag{27}$$

This function is particularly intuitive to plan with when

$$b = h\sqrt{r_x^2 + r_y^2}, \tag{28}$$

where h defines the "curviness" of the trajectory. For example, $h = 0.01$ always leads to a low arc for any target position, while $h = 0.5$ always leads to a fairly high arc.

6.6. Hitting the Target with (Almost) Minimal Effort. A trajectory can also be planned to hit a target at $[r_x, r_y]$ with the smallest initial speed (and thus the least amount of energy) possible. Planning a trajectory this way requires finding the positive time-to-target t_r which solves $\partial(v_{0,x}^2 + v_{0,y}^2)/\partial t_r = 0$, where $v_{0,x}$ and $v_{0,y}$ are defined by (21) and (22), respectively. This equation can be expanded into the following form:

$$\left(k^2 v_\infty^2\right) t_r^4 + \left(k^2 v_\infty r_y\right) t_r^3 - k\left(r_x^2 + r_y^2\right) t_r - \left(r_x^2 + r_y^2\right) = 0. \tag{29}$$

Like all quartic equations, solving this in a closed form is possible but difficult to do robustly [10]. In practice, quartic equations are typically solved for any or all of their roots by generic iterative root solvers [11]. But by exploiting domain-specific knowledge, it is also possible to implement a specialized iterative solver for (29) that is guaranteed to efficiently converge to the right root directly. One possible implementation is presented as a C++ function called *GetTimeToTargetRWithMinimalInitialSpeed*() in Algorithm 1. There, the equation is first transformed into an equivalent but more well-behaved strictly convex quartic function for which a conservative initial guess for t (or rather, u) is calculated,

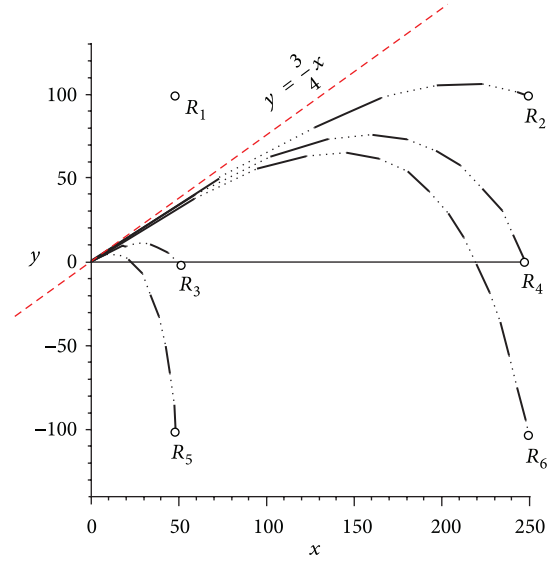

Figure 12: Planning trajectories through six different targets, all starting with the slope 3/4.

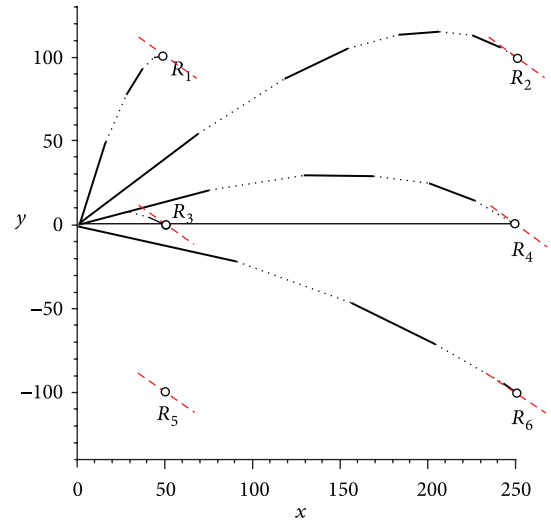

Figure 13: Planning trajectories through six different targets, all ending at the slope −3/4.

which is then refined using multiple (but typically less than a total of six) modified or normal conservative Newton's method iterations. This implementation has been carefully crafted with both robustness and efficiency in mind. As with any implementation, numeric precision can become an issue when using extreme values, but results for practical ranges are typically within a few *float* epsilons of the exact value. See the comments in the implementation itself for more details.

Alternatively, when only a rough approximation of the minimal effort solution is needed, the simpler "curviness" planner from (27) and (28) could be used with $h = 1/4$. This approximation is fairly accurate for larger values of v_∞, as it actually converges perfectly to the exact solution when v_∞ goes to infinity. But for high friction scenarios, the difference

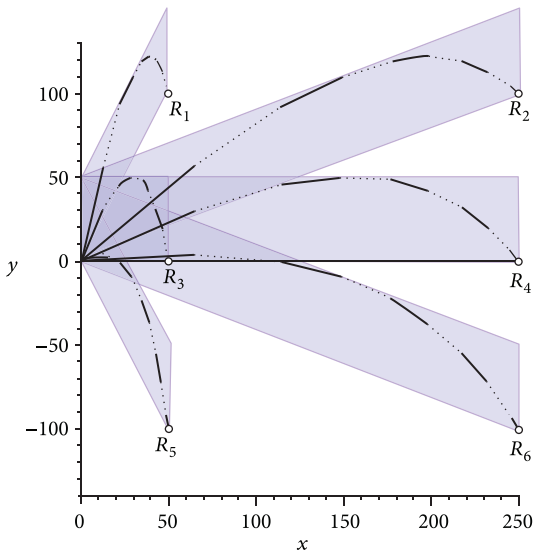

FIGURE 14: Planning trajectories through six different Rs, all with an arc height of 50.

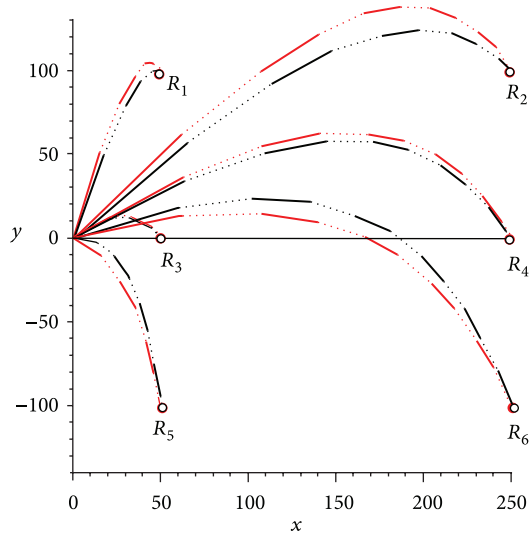

FIGURE 15: Planning trajectories through six different Rs while minimizing the initial velocity using the exact method (in black) and the approximate $h = 1/4$ method (in red) in a medium-fiction scenario.

between the exact method and the approximation becomes quite noticeable. To get an idea of the size of the error for a medium friction scenario, the trajectories resulting from the exact method and the approximation are shown side by side in Figure 15 for the case of $g = 9.81\ \text{m/s}^2$ and $v_\infty = 25\ \text{m/s}$.

6.7. Hitting a Target Given the Initial Speed. The last planning algorithm covered in this paper solves for the time t_r required to hit the target $[r_x, r_y]$ given the projectile's exact initial speed s. The solution is found by solving $v_{0,x}^2 + v_{0,y}^2 = s^2$ for the positive time t_r to position $[r_x, r_y]$, where $v_{0,x}$ and $v_{0,y}$ are once again defined by (21) and (22), respectively.

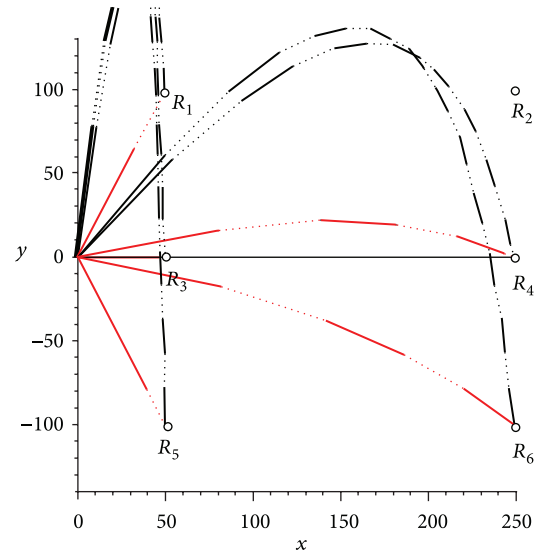

FIGURE 16: Planning trajectories through six different Rs, all starting at 100 m/s. Each of the five reachable targets has a trajectory with a (red) low arc and a (black) high arc.

This particular equation may be expanded into the following quartic function:

$$\left(k^2 v_\infty^2\right) t_r^4 + \left(2k^2 v_\infty r_y\right) t_r^3 + \left(k^2 r_x^2 + k^2 r_y^2 + 2k v_\infty r_y - s^2\right) t_r^2 + 2k\left(r_x^2 + r_y^2\right) t_r + \left(r_x^2 + r_y^2\right) = 0. \tag{30}$$

Note that when s is smaller than the minimal initial speed to hit $[r_x, r_y]$ (i.e., the sought solution of (29)), the problem has no valid solutions. But when s is larger than that, it will always have exactly two valid solutions. In that case, the smaller of the two t_r values represents the time to hit the target with a low arc, while the larger t_r root is the solution for a high arc. See Figure 16 for an example.

Using a similar approach as in the previous section, a specialized iterative solver can be implemented to solve this particular quartic function. The C++ function *GetTimeToTargetRGivenInitialSpeedS*() in Algorithm 2 solves the equation for either the high or low arc root and returns the resulting t_r or returns 0 if no solution exists. This implementation has also been written with robustness, efficiency, and accuracy for a wide range of parameters in mind. Tests showed that the procedure typically requires about six (modified) Newton's method iterations in total to converge to almost full *float* precision.

7. Future work

As already hinted at in Section 4, planning to hit moving targets can sometimes be done using a feedback loop between a target position prediction formula and a planner for a static target, together converging to a solution over multiple iterations. More research is necessary to explore the exact

boundary conditions for this convergence to occur or alternatively to look for ways to solve these problems analytically.

Additionally, it is likely that the model presented in this paper can also be used for efficient exact collision detection between a trajectory and an arbitrary polygonal mesh by testing the trajectory in principal space against the intersection of the mesh and the principal plane. This algorithm might even be combined with the planners from Sections 6.1 and 6.2 to allow for efficient planning of the most optimal trajectory above or below a given polygonal mesh, respectively. These possibilities have not been investigated in depth for this paper but might be covered in future work.

8. Conclusion

A novel analytic approximation of ballistic trajectories with air resistance has been presented that was designed to balance physical accuracy and performance in a way that makes sense in the field of computer games.

The approximation's linearity in velocity has been used to define a special principal frame of reference, which makes it possible to always work with these trajectories in a simplified 2D space, even though the original problem can be in 3D with wind coming from any direction. The combined result is that the proposed model is able to produce trajectories that are complex enough to be physically plausible, while keeping the math simple enough to also allow for many different ways of efficient trajectory planning that otherwise might be too impractical for use in computer games.

Conflict of Interests

The author declares that there is no conflict of interests regarding the publication of this paper.

References

[1] G. M. Gregorek, *Aerodynamic Drag of Model Rockets*, Estes Industries, Penrose, Colo, USA, 1970.

[2] R. G. Watts and R. Ferrer, "The lateral force on a spinning sphere: aerodynamics of a curveball," *American Journal of Physics*, vol. 55, no. 1, pp. 40–44, 1987.

[3] H. D. Curtis, *Orbital Mechanics for Engineering Students*, ch 1–5, Elsevier Butterworth-Heinemann, Burlington, Mass, USA, 1st edition, 2005.

[4] S. T. Thornton and J. B. Marion, *Classical Dynamics of Particles and Systems*, Brooks/Cole, Belmont, Calif, USA, 2005.

[5] P. A. Karkantzakos, "Time of flight and range of the motion of a projectile in a constant gravitational field under the influence of a retarding force proportional to the velocity," *Journal of Engineering Science and Technology Review*, vol. 2, no. 1, pp. 76–81, 2009.

[6] R. D. H. Warburton, J. Wang, and J. Burgdöfer, "Analytic approximations of projectile motion with quadratic air resistance," *Journal Service Science & Management*, no. 3, pp. 98–105, 2010.

[7] P. S. Chudinov, "Approximate analytical investigation of projectile motion in a medium with quadratic drag force," *International Journal of Sports Science and Engineering*, vol. 5, no. 1, pp. 27–42, 2011.

[8] G. W. Parker, "Projectile motion with air resistance quadratic in the speed," *American Journal of Physics*, vol. 45, no. 7, pp. 606–610, 1997.

[9] D. M. Bourg, *Physics for Game Developers*, O'Reilly Media, 2002.

[10] D. Herbison-Evans, "Solving quartics and cubics for graphics," Tech. Rep. TR94-487, Basser Department of Computer Science, University of Sydney, 1994.

[11] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Ch 9, Cambridge University Press, 2nd edition, 1992.