The Forward Problem - Where Does the Arrow Go?

The Goal & The Challenge

The fundamental problem is that you want to hit a distant target with an arrow. You can't just aim directly at it because of gravity. The arrow travels in a parabolic arc, meaning you have to aim *above* your target to compensate for the drop. The critical question is, "How much above?"

To answer this, we need to predict the arrow's flight path. The AimBow mod does this by creating a simulation. It builds a mathematical model of the arrow and calculates its position and velocity at small time intervals, or "ticks," to trace its entire trajectory. This is a classic physics problem of **projectile motion**.

Setting the Stage - The `RayData` Object

To simulate the arrow, you first need a way to describe its state at any given moment. The essential properties are its **position** (its x, y, z coordinates) and its **velocity** (the direction and speed it's traveling).

The mod accomplishes this with a "ghost" arrow, an abstract representation called `RayData`. This object doesn't exist in the game world but follows all the same physical rules.

Let's look at the code that defines this state, starting with the `TickingEntity.java` class, which is the base for `RayData`:

File: `src/main/java/net/famzangl/minecraft/aimbow/aiming/TickingEntity.java`

Java

```java
public class TickingEntity {
      public double posX, posY, posZ;
      public double motionX, motionY, motionZ;
    // ... other variables
}
```

Here, `posX`, `posY`, and `posZ` are the 3D coordinates. The key part is `motionX`, `motionY`, and `motionZ`. This is the **velocity vector**. It's a set of three numbers that simultaneously defines:

1. The **direction** of travel in 3D space.
2. The **speed** of travel in that direction.

By tracking these six variables, you have a complete snapshot of the arrow's state at any instant.

The Launch - Turning Player Aim into Initial Velocity

Now that we have our abstract arrow, we need to give it an initial push. The arrow's starting velocity is determined by two factors: the **power** from the bow charge and the **direction** the player is aiming.

First, let's determine the power. This is calculated in the `BowColissionSolver`.

File: `src/main/java/net/famzangl/minecraft/aimbow/aiming/Bow/BowColissionSolver.java`

Java

```
@Override
public RayData generateRayData() {
    int useDuration =
Minecraft.getMinecraft().thePlayer.getItemInUseDuration();
    // ...
    float drawTime = Math.min(useDuration, 20) / 20.0f;
    force = 2 * drawTime * drawTime * drawTime;
    // ...
    return new BowRayData(force);
}
```

Let's break this down:

- `useDuration` is the number of ticks the player has been holding the right mouse button.
- `Math.min(useDuration, 20)` caps this value at 20 ticks (1 second), which is the time for a full draw in Minecraft.
- Dividing by `20.0f` normalizes `drawTime` to a value between 0.0 (no charge) and 1.0 (full charge).
- The crucial line is `force = 2 * drawTime * drawTime * drawTime;`. This is a **cubic** relationship, not linear.
    - At a half charge (`drawTime` = 0.5), the force isn't half; it's `2 * 0.5^3 = 0.25`.
    - At full charge (drawTime = 1), the force is 2 * 1^3 = 2.
      This cubic curve means the power ramps up slowly at first and then very rapidly at the end, which mimics the feel of drawing a real bow. This force variable is the scalar magnitude of our initial velocity—it's the "speed" component.

Next, we need the direction. This is handled inside the `BowRayData` object itself.

File: `src/main/java/net/famzangl/minecraft/aimbow/aiming/Bow/BowRayData.java`

Java

```
public void shoot() {
    // ...
    double motionX = -MathHelper.sin(this.rotationYaw / 180.0F *
(float) Math.PI)
                    * MathHelper.cos(this.rotationPitch / 180.0F *
(float) Math.PI);
    double motionZ = MathHelper.cos(this.rotationYaw / 180.0F *
(float) Math.PI)
                    * MathHelper.cos(this.rotationPitch / 180.0F *
(float) Math.PI);
    double motionY = (-MathHelper.sin(this.rotationPitch / 180.0F *
(float) Math.PI));
    this.setThrowableHeading(motionX, motionY, motionZ, force *
1.5F, 0);
}
```

This is pure trigonometry. It converts the player's viewing angles (`rotationYaw` and `rotationPitch`) from spherical coordinates into a 3D Cartesian vector (`motionX`, `motionY`, `motionZ`). This vector has a length of 1; it's a "unit vector" that represents only direction.

Finally, the call to `setThrowableHeading` combines the two. It takes the direction vector and scales it by the `force` we calculated earlier (multiplied by a constant `1.5F` for bows). Now our ghost arrow has its complete initial velocity vector and is ready to be simulated.

The Simulation - One Tick at a Time

Calculating the arrow's final landing spot with a single equation is extremely difficult because of factors like air resistance. The mod uses a much more practical approach: it simulates the flight tick by tick. This numerical method is a form of **Euler integration**.

The core of this simulation is in the `moveTick` method of the `RayData` class.

File: `src/main/java/net/famzangl/minecraft/aimbow/aiming/RayData.java`

```java
@Override
public void moveTick() {
    super.moveTick();
    // ... rotation calculations ...
    float f3 = 0.99F; // Air resistance factor
    float f1 = getGravity();

    this.motionX *= f3;
    this.motionY *= f3;
    this.motionZ *= f3;
    this.motionY -= f1;
    this.setPosition(this.posX, this.posY, this.posZ);

    trajectory.add(new Vec3(this.posX, this.posY, this.posZ));
}
```

Three critical physics principles are applied here every tick:

1. **Gravity:** The line `this.motionY -= f1;` is where gravity is applied. For a bow, `getGravity()` returns `0.05f`. This means every tick, the arrow's vertical velocity is reduced by a constant amount. This relentless downward acceleration is what creates the parabolic arc.
2. **Air Resistance (Drag):** The lines `this.motionX *= f3;`, `this.motionY *= f3;`, and `this.motionZ *= f3;` model drag. With `f3 = 0.99F`, the arrow loses 1% of its velocity in all directions each tick. This is why the arrow eventually slows down and doesn't fly forever.
3. Position Update: The `moveTick` method in the parent `TickingEntity` class performs the final step:
   `this.posX += this.motionX;`
   This is the Euler integration step. The new position is simply the old position plus the velocity vector for that tick.

By calling this `moveTick` method in a loop, the mod traces the arrow's entire path. Each calculated position is added to the `trajectory` list, which is then rendered on the screen as the visible line.

The Reverse Problem - How Do We Hit What We Aim At?

The New Challenge - Finding the Right Angle

Now we get to the really clever part: the auto-aim. The problem is now inverted. We know the destination (the target entity), and we need to find the initial launch angle (`pitch`) that will get the arrow there.

A direct analytical solution is extremely difficult. Instead of trying to solve one complex equation, the mod uses an **iterative algorithm**. It makes an intelligent guess for the angle, simulates the shot to see where it lands, and then uses that result to make a better guess, repeating until it finds the perfect angle.

The "Smart Guesser" - The `ReverseBowSolver`

The class that performs this magic is the `ReverseBowSolver`. It uses a highly efficient algorithm called the **bisection method** (a type of binary search).

Imagine you're guessing a number between 1 and 100. You guess 50. If you're told "too high," you've just eliminated half of all possibilities. Your next guess is 25. By repeatedly guessing the midpoint and halving the search space, you find the answer very quickly. The mod does this, but for angles.

File: `src/main/java/net/famzangl/minecraft/aimbow/aiming/Bow/ReverseBowSolver.java`

Java

```
private float getYForTarget(float dHor, float dVert) {
    float maxVert = 0.9f, minVert = -0.9f;
    for (int attempts = 0; attempts < 50; attempts++) {
        float vert = (maxVert + minVert) / 2;
        // ...
        float newY = getYAtDistance(hor * velocity, vert * velocity,
dHor);
        if (Float.isNaN(newY)) {
            return 0;
        } else if (newY > dVert) {
```

```
            maxVert = vert;
        } else {
            minVert = vert;
        }
    }
    float res = (maxVert + minVert) / 2;
    return res;
}
```

Let's dissect this:

- `float maxVert = 0.9f, minVert = -0.9f;` defines the initial search space. We know the vertical component of our launch direction must be between aiming mostly down and mostly up.
- The `for` loop runs 50 times to ensure high precision.
- `float vert = (maxVert + minVert) / 2;` is the **guess**. It picks an angle exactly in the middle of the current possible range.
- `float newY = getYAtDistance(...)` is the **check**. This method runs a mini-simulation using the same physics from Act 1. It calculates the arrow's height (`newY`) when it reaches the target's horizontal distance (`dHor`).
- The `if/else` block is where the search space is **refined**:
    - If `newY > dVert`, the shot went too high. The guessed angle was too large. Therefore, the new maximum possible angle (`maxVert`) becomes our guess. We have just eliminated the entire top half of the search range.
    - Otherwise, the shot was too low. The new minimum possible angle (`minVert`) becomes our guess, eliminating the bottom half.

After 50 iterations of halving the search space, the difference between `minVert` and `maxVert` is practically zero, and their average is the extremely precise vertical launch vector needed.

Putting It All Together - The Final Aim

The `ReverseBowSolver` has given us the perfect 3D vector to aim in. The final step is to translate this back into the yaw and pitch angles that the player's view uses.

File: `src/main/java/net/famzangl/minecraft/aimbow/AimbowGui.java`

```java
private void adjustPlayerLook(Vec3 lookDir) {
    // ...
    float yaw = (float) Math.toDegrees(Math.atan2(dz, dx)) - 90f;
    float pitch = (float) -Math.toDegrees(Math.atan2(dy,
Math.sqrt(dx*dx + dz*dz)));
    // ...
    mc.thePlayer.setAngles(yawDiff/0.15f, -pitchDiff/0.15f);
}
```

This is the reverse of the trigonometry from the start. It uses the `atan2` (arc-tangent) function to convert the `(dx, dy, dz)` vector back into `yaw` and `pitch` angles. The mod then smoothly adjusts the player's view to match this perfect aim.

By combining a step-by-step physics simulation with an elegant iterative search algorithm, the AimBow mod provides a powerful and mathematically sound tool for archers.