All successful targeting and shooting of enemy robots requires an algorithm to fire bullets at the place where you predict that an enemy robot will be at a future point in time. This algorithm can be used for linear, circular, and oscillating predictive targeting. And if you have a function that returns the position of the enemy at a future point in time, you can use the algorithm to calculate the impact point, the firing angle, the impact position, and the impact time.

This algorithm implements the secant method to numerically solve the impact time. Once this impact time is known, our predictive function obtains the impact position. Then we fire at that position.

The Intercept class shown in Listing 1 assumes that the robot is traveling in a straight line from its current position at its current velocity.

**Listing 1. Using the Intercept class**

```java
public class Intercept {
    public Coordinate impactPoint = new Coordinate(0,0);
    public double bulletHeading_deg;

    protected Coordinate bulletStartingPoint = new Coordinate();
    protected Coordinate targetStartingPoint = new Coordinate();
    public double targetHeading;
    public double targetVelocity;
    public double bulletPower;
    public double angleThreshold;
    public double distance;

    protected double impactTime;
    protected double angularVelocity_rad_per_sec;

    public void calculate ( // Initial bullet position x coordinate
double xb,
    // Initial bullet position y coordinate
double yb,
```
// Initial target position x coordinate
double xt,
// Initial target position y coordinate
double yt,
// Target heading
double tHeading,
// Target velocity
double vt,
// Power of the bullet that we will be firing
double bPower,
// Angular velocity of the target
double angularVelocity_deg_per_sec
)
{
    angularVelocity_rad_per_sec = Math.toRadians(angularVelocity_deg_per_sec);
    bulletStartingPoint.set(xb, yb);
    targetStartingPoint.set(xt, yt);
    targetHeading = tHeading;
    targetVelocity = vt;
    bulletPower = bPower;
    double vb = 20-3*bulletPower;
    double dx,dy;
    // Start with initial guesses at 10 and 20 ticks
    impactTime = getImpactTime(10, 20, 0.01);
    impactPoint = getEstimatedPosition(impactTime);
    dx = (impactPoint.x - bulletStartingPoint.x);
    dy = (impactPoint.y - bulletStartingPoint.y);
    distance = Math.sqrt(dx*dx + dy*dy);
    bulletHeading_deg = Math.toDegrees(Math.atan2(dx,dy));
    angleThreshold = Math.toDegrees(Math.atan(ROBOT_RADIUS/distance));
}

protected Coordinate getEstimatedPosition(double time) {
    double x = targetStartingPoint.x +
        targetVelocity * time * Math.sin(Math.toRadians(targetHeading));
    double y = targetStartingPoint.y +
        targetVelocity * time * Math.cos(Math.toRadians(targetHeading));
    return new Coordinate(x,y);
}

private double f(double time) {
    double vb = 20-3*bulletPower;
    Coordinate targetPosition = getEstimatedPosition(time);
    double dX = (targetPosition.x - bulletStartingPoint.x);
    double dY = (targetPosition.y - bulletStartingPoint.y);
    return Math.sqrt(dX*dX + dY*dY) - vb * time;
}

private double getImpactTime(double t0,
    double t1, double accuracy) {
    double X = t1;
    double lastX = t0;
    int iterationCount = 0;
double lastfX = f(lastX);

while ((Math.abs(X - lastX) >= accuracy) &&
   (iterationCount < 15)) {

    iterationCount++;
    double fX = f(X);

    if ((fX-lastfX) == 0.0) break;

double nextX = X - fX*(X-lastX)/(fX-lastfX);
lastX = X;
X = nextX;
lastfX = fX;
}
return X;
}

The great thing about the Intercept class is that it can be easily reused to calculate the firing angle for circular motion. To do this, write a CircularIntercept class that inherits from the Intercept class, and overwrite the getEstimatedPosition() method. Listing 2 shows the code for the CircularIntercept class:

**Listing 2. CircularIntercept class**

```java
public class CircularIntercept extends Intercept {
  protected Coordinate getEstimatedPosition(double time) {
    if (Math.abs(angularVelocity_rad_per_sec) <= Math.toRadians(0.1)) {
      return super.getEstimatedPosition(time);
    }

    double initialTargetHeading = Math.toRadians(targetHeading);
    double finalTargetHeading = initialTargetHeading +
      angularVelocity_rad_per_sec * time;
    double x = targetStartingPoint.x - targetVelocity /
      angularVelocity_rad_per_sec *(Math.cos(finalTargetHeading) -
      Math.cos(initialTargetHeading));
    double y = targetStartingPoint.y - targetVelocity /
      angularVelocity_rad_per_sec *
      (Math.sin(initialTargetHeading) -
      Math.sin(finalTargetHeading));
    return new Coordinate(x,y);
  }
}
```

Listing 3 shows an example of using the Intercept class. It assumes that we calculated the current position, heading, and velocity of the target, as well as the power of the bullet that we will be firing.

**Listing 3. Using the Intercept class**

```java
Intercept intercept = new Intercept();
intercept.calculate(
    ourRobotPositionX,
    ourRobotPositionY,
    currentTargetPositionX,
    currentTargetPositionY,
    targetVelocity,
    angularVelocity_rad_per_sec,
    targetStartingPoint.x,
    targetStartingPoint.y,
    Math.toRadians(0.1));
```
currentTargetPositionY,
currentTargetHeading_deg,
currentTargetVelocity,
bulletPower,
0 // Angular velocity
);

// Helper function that converts any angle into
// an angle between +180 and -180 degrees.
double turnAngle = normalRelativeAngle
(intercept.bulletHeading_deg - robot.getGunHeading());

// Move gun to target angle
robot.setTurnGunRight(turnAngle);

if (Math.abs(turnAngle) <= intercept.angleThreshold) {
  // Ensure that the gun is pointing at the correct angle
  if (
    (intercept.impactPoint.x > 0) &&
    (intercept.impactPoint.x < getBattleFieldWidth()) &&
    (intercept.impactPoint.y > 0) &&
    (intercept.impactPoint.y < getBattleFieldHeight())
  ){
    // Ensure that the predicted impact point is within
    // the battlefield
    fire(bulletPower);
  }
}

This firing strategy has proven very successful in both of my robots (JollyNinja and MadHatter). Encapsulating the intercept in its own class and subclassing it for different prediction algorithms allows me to usually hit other robots more than they hit me.
Related topics

- Read all of the *Secrets from the Robocode masters*. This page will be updated as new tips become available.
- Robocode’s creator, Mathew Nelson, maintains the [official Robocode site](https://developerWorks.com). This should be the first stop for anyone serious about Robocode.
- "Rock 'em, sock 'em Robocode" ([developerWorks](https://developerWorks.com), January 2002) disarms Robocode and starts you on your way to building your own customized lean, mean, fighting machine.
- In "Rock 'em, sock 'em Robocode: Round 2" ([developerWorks](https://developerWorks.com), May 2002), Sing Li looks at advanced robot construction and team play.
- New to Java? Check out "Introduction to Java programming" ([developerWorks](https://developerWorks.com), November 2004), a tutorial that steps you through the fundamentals of Java language programming.

© Copyright IBM Corporation 2002

[Trademarks](https://developerWorks.ibm.com/ibm/trademarks/)