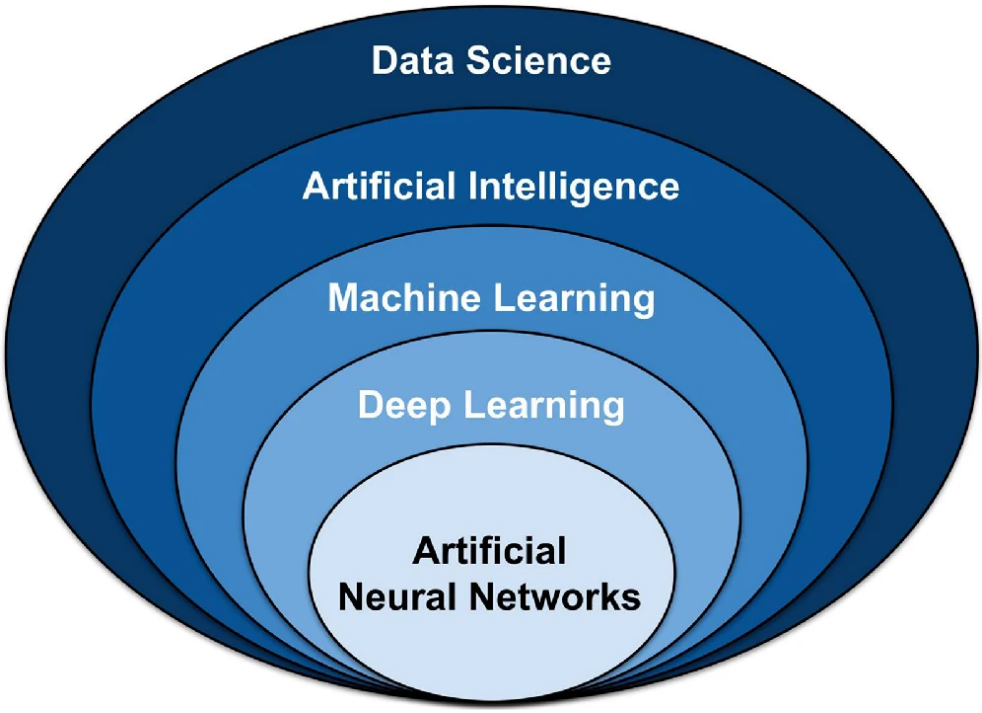
Robocode Tank Royale.



1. **Preparación del entorno**
2. **Mi primer Bot**
   2. 1. Proyecto IntelliJ
   2. 2. Configuración del servidor (GUI) y el proyecto IntelliJ para ejecutar en local o remoto
3. **¿Cómo mejoro mi Bot?**
   3. 1. Conociendo el campo de batalla
      3. 1. 1. Eventos própios
   3. 2. Técnicas de escaneo (Radar)
   3. 3. Técnicas de desplazamiento (Movimiento)
      3. 3. 1. Detección de colisiones más rápida
      3. 3. 2. Pensamiento lateral
      3. 3. 3. ¿Hacia adelante o hacia atrás?
      3. 3. 4. Cambiar de dirección
      3. 3. 5. ¿Bailamos?
      3. 3. 6. Evitando las paredes
      3. 3. 7. Bot multimodo
   3. 4. Técnicas de ataque (Disparo)
      3. 4. 1. Técnicas básicas
      3. 4. 2. Fórmula de cálculo de potencia de fuego
      3. 4. 3. Evitar disparos prematuros
      3. 4. 4. Apuntando de manera predictiva
      3. 4. 5. Obteniendo futuras coordenadas X,Y
      3. 4. 6. Girando el arma al punto previsto
4. **Investigación y desarrollo propio**
5. **Fuentes de información**

1. Preparación del entorno

- Al menos java 11 (Yo estoy usando Java 19 y la versión 0.19.3 de Tank Royale)

```
java -version
```

- Descargar la última versión desde releases: https://github.com/robocode-dev/tank-royale/releases
- Ejecutar el `jar` descargado:

```
java -jar robocode-tankroyale-gui-x.y.z.jar
```

- Descarga y descomprime los Bots de ejemplo: https://github.com/robocode-dev/tank-royale/releases
- Configura la gui para acceder a la carpeta de robots: `Config` -> `Bot Root Directories` (del menú)
- [opcional] Añadir sonidos al gui. Descarga y descomprime la carpeta `sounds.zip` de: https://github.com/robocode-dev/sounds/releases

```
[directorio padre]
├── robocode-tankroyale-gui-x.y.z.jar
└── sounds/  <-- carpeta de sonidos
    ├── bots_collision.wav
    ...
    └── wall_collision.wav
```

- Necesitaras IntelliJ para poner todo en funcionamiento y para programar tu Bot.

Ampliación Juega un poco por tu cuenta con el entorno GUI, explora las opciones, tipos de batallas, crea alguna ronda de las mismas, inicializa Bots, añádelos, juega con la velocidad de juego, etc.

2. Mi primer Bot

2.1. Proyecto IntelliJ

Para acelerar el proceso, he preparado un proyecto en IntelliJ con toda la arquitectura básica que necesitará tu Bot. Descarga y descomprime esta carpeta y abre el proyecto con el IDE IntelliJ.

También necesitaras la libreria (API) de Robocode Tank Royale que puedes descargar desde aquí: https://github.com/robocode-dev/tank-royale/releases. Descarga el archivo: `robocode-tankroyale-bot-api-x.y.z.jar`

La estructura debería quedar de la siguiente manera:

```
[directorio padre]
├── lib
|   └──robocode-tankroyale-gui-x.y.z.jar
└── IABDBot  <-- Proyecto de IntelliJ
    ├── src
    ...
    └── IABDBot.iml
```

Importante Asegurate de entender el funcionamiento del Bot IABDBot, tiene comentarios donde se explican partes importante del Bot y que daremos por conocidas en los siguientes puntos.

Es muy importante que entiendas la diferencia entre movimientos bloqueantes y simultáneos, así como las condiciones y los eventos disponibles.

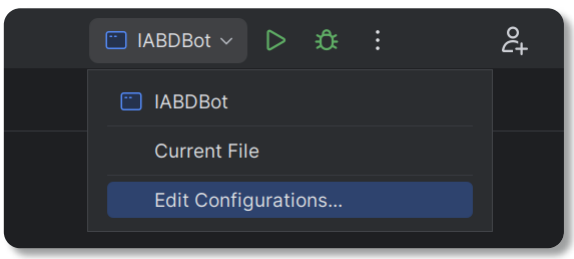
2.2. Configuración del servidor (GUI) y el proyecto IntelliJ para ejecutar en local o remoto

El servidor permite conexiones en remoto/local a través de un password que se genera automáticamente la primera vez que lanzas la GUI. Este password lo puedes encontrar/modificar en el archivo `server.properties`, en el ejemplo siguiente la clave de acceso es **CEIABDEPM2023**

```
bots-secrets=CEIABDEPM2023
[...]
```

Otra información que necesitaras será la IP del servidor al que quieres conectar, normalmente será `localhost` (tu host local), y cuando sea necesario hacer pruebas el profesor te facilitará su IP.

Ahora solo queda configurar nuestro proyecto de IntelliJ para configurar estas variables en el momento de ejecutar el Bot. Accede a la configuración de las ejecuciones en la parte superior derecha (una vez abierta la clase `IABDBot.java`):



Puedes ver que en el apartado `Environment Variables` tienes el siguiente valor:

```
SERVER_SECRET=CEIABDEPM2023;SERVER_URL=ws://localhost:7654
```

Como puedes imaginar **SERVER_SECRET** hace referencia a la clave del servidor, y **SERVER_URL** a la dirección del servidor y es donde deberás reemplazar (según el caso) localhost por la IP que te indique el profesor. Así podrás ejecutar tu Bot directamente desde IntelliJ y aparecerá en el Servidor para poder añadirlo a las batallas.

Importante Ojo, el servidor debe estar inicializado antes de lanzar el bot de lo contrario obendrás un error similar a este:

```
Exception in thread "main" dev.robocode.tankroyale.botapi.BotException: Could not create web socket for URL: ws://localhost:7654
    at dev.robocode.tankroyale.botapi.internal.BaseBotInternals.connect(BaseBotInternals.java:268)
    at dev.robocode.tankroyale.botapi.internal.BaseBotInternals.start(BaseBotInternals.java:254)
    at dev.robocode.tankroyale.botapi.BaseBot.start(BaseBot.java:114)
    at IABDBot.main(IABDBot.java:11)

Process finished with exit code 1
```

Si el Bot encuentra el servidor y la contraseña es correcta deberia aparecer esto al inicializarlo en IntelliJ:

```
Connected to: ws://localhost:7654
```

Ampliación Prueba por tu cuenta mezclando en las batallas Bots de ejemplo, con tu Bot o incluso el de algún compañero/a.

3. ¿Cómo mejoro mi Bot?

Dividiremos todo lo que debemos conocer en 4 apartados:

3.1. Conociendo el campo de batalla

Sigue atentamente la documentación de RCTR sobre la **anatomía** de los Bots

Puntos importantes:

- Si el radar no se mueve, no detecta
- Inicialmente el robot, el cañón y el radar se mueven conjuntamente (pero se puede cambiar)
- El Bot se simplifica a un cuadrado de 36x36 unidades.
- Para las colisiones (con Bots o paredes) se simula como un círculo de 18 unidades de radio.

A continuación revisa las **coordenadas y ángulos**

Puntos importantes:

- Este apartado es totalmente diferente al de Robocode Original.

Estudia también las **físicas**

Puntos importantes:

- Se frena más rápido que se acelera.
- Cuanto más rápido vas, más lento giras.
- El cañón gira máximo 20º por turno.
- El radar gira máximo 45º por turno.
- La potencia de tiro influye en el daño provocado y los puntos conseguidos, pero también en el calentamiento del cañón.
- El atropello da más puntos que los disparos (pero te resta energia)
- Si chocas con una pared, pierdes puntos.
- La energía que te sobra en una ronda no da puntos (modo kamikaze con el último robot?)

Por último te en cuenta las **puntuaciones**

Puntos importantes:

- Consigues puntos si:
    - tu disparo golpea a otro Bot (sino, te resta)
    - eres el que mata al Bot
    - cada vez que otro Bot muere, pero tu sigues vivo
    - eres el último robot vivo
    - atropellas a otro Bot
    - si matas por atropello a otro Bot
- El último Bot que quede vivo no tiene porqué ser el ganador.

3.1.1. Eventos própios

Definimos una condición, y cuando esta se cumple se dispara un evento:

```
    // Esta condición se dispara cuando el bot completa su giro
    public static class TurnCompleteCondition extends Condition {

        private final IBot bot;

        public TurnCompleteCondition(IBot bot) {
            this.bot = bot;
        }

        @Override
        public boolean test() {
```

```
            // El método test() es el que debemos reescribir para definir el resultado de
nuestra condición.
            return bot.getTurnRemaining() == 0;
        }
    }
```

Podriamos usar esta condición en un fragmento similar a este:

```
waitFor(new TurnCompleteCondition());
```

Podriamos usar funciones Lambda de la siguiente manera:

```
waitFor(new Condition(() -> getTurnRemaining() == 0);
```

El resultado de los dos fragmentos seria el mismo.

3.2. Técnicas de escaneo (Radar)

Sentidos de nuestro Bot:

Sentido del **tacto**, tu Bot sabe cuando:

- golpea una pared (`onHitWall`),
- es alcanzado por un disparo (`onHitByBullet`),
- es alcanzado por otro Bot (`onHitBot`).

Sentido de la **vista**, tu Bot sabe cuándo ha visto otro robot, pero sólo si lo escanea (`onScannedBot`)

Otros sentidos, tu robot también sabe cuándo ha muerto (`onDeath`), cuándo ha muerto otro robot (`onRobotDeath`).

Además también es consciente de sus balas y sabe cuando una bala ha sido disparada (`onBulletFired`) ha alcanzado a un oponente (`onBulletHit`), cuando una bala golpea una pared (`onBulletWall`) o cuando una bala golpea a otra bala (`onBulletHitBullet`).

Fijar el radar en un enemigo:

- Arco de radar Estrecho
- Radar Oscilante
- Escaneo más inteligente (seguir al cercano)

3.3. Técnicas de desplazamiento (Movimiento)

3.3.1. Detección de colisiones más rápida

Podemos determinar si un punto está dentro de las 18 unidades del centro del bot. Con la fórmula:

$$d^2 = \Delta x^2 + \Delta y^2$$

Esto significa que sólo necesitamos calcular Δx^2 + Δy^2 y ver si es menor que 324 (18^2) para ver si el punto está dentro del círculo delimitador.z

3.3.2. Pensamiento lateral

Si has jugado baloncesto antes, sabes que si quieres defender a alguien que sostiene el balón, debes maximizar tu movimiento lateral enfrentándote siempre a él (de frente). Lo mismo ocurre con tu robot.

Para conseguir esta posición debemos hacer algo similar a esto:

```
setTurnLeft(e.getBearing() + 90);
```

que siempre colocará tu robot perpendicular (90 grados) a tu enemigo.

3.3.3. ¿Hacia adelante o hacia atrás?

Cuando te enfrentas a un oponente, la idea de "adelante" y "atrás" (del primer Bot) se vuelven algo obsoletas. Probablemente estés pensando más en términos de "atacar a la izquierda" o "atacar a la derecha". Para realizar un seguimiento de la dirección del movimiento, simplemente declare una variable como hicimos para oscilar el radar.

```
class MyRobot extends Bot {
    private byte moveDirection = 1;
```

luego, cuando quieras mover tu robot, simplemente puedes decir:

```
setAhead(100 * moveDirection);
```

Puedes cambiar de dirección cambiando el valor de moveDirection de 1 a -1 así:

```
moveDirection *= -1;
```

3.3.4. Cambiar de dirección

El enfoque más intuitivo para cambiar de dirección es simplemente cambiar la dirección del movimiento cada vez que golpeas una pared o golpeas a otro robot de esta manera:

```
public void onHitWall(HitWallEvent e) { moveDirection *= -1; }
public void onHitRobot(HitRobotEvent e) { moveDirection *= -1; }
```

Sin embargo, descubrirás que si haces eso, terminarás presionando obstinadamente contra un robot que te embiste desde un costado (como un perro en celo). Esto se debe a que se llama a `onHitRobot()` tantas veces que moveDirection sigue cambiando y nunca te alejas.

Un mejor enfoque es simplemente probar para ver si su robot se ha detenido. Si es así, probablemente significa que has golpeado algo y querrás cambiar de dirección. Puedes hacerlo con el código:

```
if (getVelocity() == 0)
    moveDirection *= -1;
```

Ponlo en tu método `doMove()` (o en cualquier otro lugar donde estés manejando el movimiento) y podrás manejar todos los eventos de impacto con las paredes u otros Bots.

3.3.5. ¿Bailamos?

Dando vueltas (Círculos)

Puedes rodear a tu enemigo simplemente usando las técnicas anteriores:

```
public void doMove() {
    // switch directions if we've stopped
    if (getVelocity() == 0)
        moveDirection *= -1;
    // circle our enemy
    setTurnLeft(lastScannedBearing-90);
    setAhead(1000 * moveDirection);
}
```

Objetivo: rodea a tu enemigo usando el código de movimiento anterior, como un tiburón rodeando a su presa en el agua.

Ametrallamiento

Un problema que puedes notar con el anterior tipo de movimiento es que es presa fácil para los objetivos predictivos porque sus movimientos son tan... predecibles.

Para evadir las balas de manera más efectiva, debes moverte de lado a lado o "atacar". Una buena forma de hacerlo es cambiar de dirección después de un cierto número de "tics", así:

```
public void doMove() {

    // always square off against our enemy
    setTurnLeft(lastScannedBearing-90);

    // strafe by changing direction every 20 ticks
    if (getTurnNumber() % 20 == 0) {
        moveDirection *= -1;
        setForward(1000 * moveDirection);
    }
}
```

Mira como se balancea hacia adelante y hacia atrás usando el código de movimiento anterior. Observa lo bien que esquiva las balas. Pero ojo, porque puede afectar a tu precisión de disparo.

Cada vez más cerca

Notarás que tanto el primero como este último tipo de movimientos tienen otro problema: se atascan fácilmente en las esquinas y terminan golpeándose contra las paredes. Un problema adicional es que si su enemigo está lejos, disparan mucho pero no aciertan mucho.

Para hacer que tu robot se acerque a tu enemigo, simplemente modifica el código para que se gire ligeramente hacia su enemigo, así:

```
setTurnLeft((lastScannedBearing + 90 - (15 * moveDirection)));
```

Modificando el primero conseguimos una variación que utiliza el código anterior para lanzarse en espiral hacia su enemigo.

Mientras que con el segundo que utiliza el código anterior para acercarse cada vez más. También es bastante bueno para evadir balas.

Fíjate que ninguno de los Bots anteriores queda atrapado en una esquina por mucho tiempo.

3.3.6. Evitando las paredes

Un problema con todos los Bots anteriores es que golpean mucho las paredes y golpearlas agota tu energía. Una mejor estrategia sería detenerse antes de chocar contra las paredes. ¿Pero cómo?

Agregar un evento personalizado

Lo primero que debemos hacer es decidir qué tan cerca permitiremos que nuestro robot llegue a las paredes:

```
public class WallAvoider extends Bot {
    ...
    private int wallMargin = 60;
```

A continuación, agregamos un evento personalizado que se activará cuando se cumpla una determinada condición:

```
// Don't get too close to the walls
addCustomEvent(new Condition("TooCloseToWalls") {
    public boolean test() {
        // El método test() es el que debemos reescribir para definir el resultado de
nuestra condición.
        return (
            // we're too close to the left wall
            (getX() <= wallMargin ||
             // or we're too close to the right wall
             getX() >= getArenaWidth() - wallMargin ||
             // or we're too close to the bottom wall
             getY() <= wallMargin ||
             // or we're too close to the top wall
             getY() >= getArenaHeight() - wallMargin)
        );
    }
});
```

Ten en cuenta que estamos creando una clase interna anónima con esta llamada. Necesitamos hacer override del método `test()` para devolver un valor booleano cuando ocurra nuestro evento personalizado.

Gestionar el evento personalizado

Lo siguiente que debemos hacer es manejar el evento, que se puede hacer así:

```
    public void onCustomEvent(CustomEvent e) {
        if (e.getCondition().getName().equals("TooCloseToWalls")) {
            // switch directions and move away
            moveDirection *= -1;
            setForward(100 * moveDirection);
        }
    }
```

Sin embargo, el problema con ese enfoque es que este evento podría dispararse una y otra vez, provocando que cambiemos rápidamente de un lado a otro, sin alejarnos nunca. O si ya aparecemos cerca de la pared quedamos atrapados.

Para evitar este "sacudida de la muerte" deberíamos tener una variable que indique que estamos manejando el evento. Podemos declarar otro así:

```
public class WallAvoider extends Bot {
    ...
    private int tooCloseToWall = 0;
```

Luego maneje el evento de manera un poco más inteligente:

```
public void onCustomEvent(CustomEvent e) {
    if (e.getCondition().getName().equals("TooCloseToWalls")) {
        if (tooCloseToWall <= 0) {
            // Si no estábamos ya cerca de las paredes, ahora lo estamos
            tooCloseToWall += wallMargin;
            setMaxSpeed(0); // Para!!!
        }
    }
}
```

Manejo de los dos modos

Hay dos últimos problemas que debemos resolver. En primer lugar, tenemos un método `doMove()` donde colocamos todo nuestro código de movimiento normal. Si estamos tratando de alejarnos de una pared, no queremos que se llame a nuestro código de movimiento normal, creando (una vez más) el "sacudida de la muerte". En segundo lugar, queremos volver eventualmente al movimiento "normal", por lo que eventualmente deberíamos tener el "tiempo de espera" de la variable `tooCloseToWall`.

Podemos resolver ambos problemas con la siguiente implementación de `doMove()`:

```
public void doMove() {
    ...

    // if we're close to the wall, eventually, we'll move away
    if (tooCloseToWall > 0) tooCloseToWall--;

    // switch directions if we've stopped
    if (getSpeed() == 0) {
        setMaxSpeed(8);
        moveDirection *= -1;
        setForward(1000 * moveDirection);
    }
}
```

Con todo el código anterior evitamos chocar contra las paredes. Observa cómo se desliza suavemente hacia los lados pero nunca (bueno, rara vez) los golpea.

3.3.7. Bot multimodo

Además de los colores que elijas, la mayor parte de la personalidad de tu robot está en su código de movimiento. Por otra parte, situaciones diferentes requieren tácticas diferentes. Usando el ejemplo de evitar paredes, es posible que desees codificar tu bot para que cambie los "modos" según ciertos criterios. Podrias pensar en algo como esto:

```
public class MultiModeBot extends Bot {
    private final static int MODO_RONDAR=0;
    private final static int MODO_ESQUIVAR=1;
    private final static int MODO_ASESINO=2;
    private int modoRobot=MODO_RONDAR;
    ...
}

public void onRobotDeath(RobotDeathEvent e) {
    ...
    if (getEnemyCount() > 10) {
        // Un gran número de enemigos sugiere movimientos fluidos
        modoRobot=MODO_RONDAR;
    } else if (getEnemyCount() > 1) {
        // esquivar es la mejor táctica para grupos pequeños
        modoRobot=MODO_ESQUIVAR;
    } else if (getEnemyCount() == 1) {
        // Si solo queda un robot, persíguelo
        modoRobot=MODO_ASESINO;
    }
    ...
}

public void doMove() {
    switch (modoRobot){
        case MODO_RONDAR:
            //ronda
            break;
        case MODO_ESQUIVAR:
            //esquiva
            break;
        case MODO_ASESINO:
            //asesina
            break;
    }
    ...
}
```

Los detalles se dejan (como siempre) como ejercicio para el alumnado.

3.4. Técnicas de ataque (Disparo)

3.4.1. Técnicas básicas

- Separa el movimiento del cañón del movimiento del Bot (`setAdjustGunForBodyTurn`)
- Técnica de apuntado básica: `setTurnGunLeft` o `setTurnGunRight` junto con `gunBearingTo`

3.4.2. Fórmula de cálculo de potencia de fuego

Otro aspecto importante al disparar es calcular la potencia de fuego de tu bala. La documentación del método fire() explica que puedes disparar una bala en el rango de `0.1` a `3.0`. Como probablemente ya habrás concluido, es una buena idea disparar balas de baja potencia cuando tu enemigo está lejos y balas de alta resistencia cuando está cerca.

```
public void smartFire(double distance){
    if (distance >200) || getEnergy() <15){
        fire(1);
    } else if (distance > 50){
        fire(2);
    } else {
        fire(3);
    }
}
```

Podrías usar una serie de declaraciones if-else-if-else para determinar la potencia de fuego, en función de si el enemigo está a 50 unidades, a 200, etc (como en el fragmento anterior). Pero tales construcciones son demasiado rígidas. Después de todo, el rango de posibles valores de potencia de fuego cae a lo largo de un continuo, no de bloques discretos. Un mejor enfoque es utilizar una fórmula. He aquí un ejemplo:

```
setFire(400/distanceTo(e.getX(), e.getY()));
```

Con esta fórmula, a medida que aumenta la distancia del enemigo, la potencia de fuego disminuye. Asimismo, a medida que el enemigo se acerca, la potencia de fuego aumenta. Los valores superiores a 3 se reducen a 3, por lo que nunca dispararemos una bala mayor que 3, pero probablemente deberíamos reducir el valor de todos modos (solo para estar seguros) de esta manera:

```
setFire(Math.max(400/distanceTo(e.getX(), e.getY()), 3));
```

3.4.3. Evitar disparos prematuros

Una situación que encontraras es que tu Bot dispare antes de haber girado el arma hacia el objetivo. Para evitar disparos prematuros, usa el método `getGunTurnRemaining()` para ver qué tan lejos está tu cañón del objetivo y no dispare hasta que esté cerca.

Además, no puedes disparar si el arma está "caliente" desde el último disparo y llamar a `fire()` o `setFire()` solo desperdiciará un turno. Podemos probar si el arma está fría llamando a `getGunHeat()`.

3.4.4. Apuntando de manera predictiva

O.. "Usar la trigonometría para impresionar a tus amigos y destruir a tus enemigos".

Si quisiéramos poder golpear a un robot que recorre las paredes siempre fallaríamos, necesitamos poder predecir dónde estará en el futuro, pero ¿cómo podemos hacerlo?

$$Distancia = Velocidad * Tiempo$$

Usando la anterior formula podemos calcular cuánto tiempo tardará una bala en llegar allí.

- **Distancia**: se puede encontrar llamando a `distanceTo(e.getX(), e.getY())`
- **Velocidad**: según la documentación de RCTR, una bala viaja a una velocidad de:

$$20 - potenciaDeFuego * 3$$

- **Tiempo**: podemos calcular el tiempo resolviendo:

$$Tiempo = \frac{Distancia}{Velocidad}$$

El siguiente código lo hace:

```
// calcular la potencia basado en la distancia
double enemyDistance = distanceTo(e.getX(), e.getY());
double firePower = Math.min(500 / enemyDistance, 3);
// calcular la velocidad de la bala
double bulletSpeed = 20 - firePower * 3;
// calculamos el tiempo que necesita la bala para impactar al enemigo
long time = (long) (enemyDistance / bulletSpeed);
```

3.4.5. Obteniendo futuras coordenadas X,Y

A continuación, debemos calcular la posición futura de nuestro enemigo:

```
// Calcular la velocidad actual de tu enemigo
double enemyVelocity = e.getSpeed();

// Calcular el desplazamiento en las coordenadas X e Y
double deltaX = enemyVelocity * Math.sin(e.getDirection()); // Componente X del desplazamiento
double deltaY = enemyVelocity * Math.cos(e.getDirection()); // Componente Y del desplazamiento

// Calcular las coordenadas futuras
double futureX = e.getX() + (deltaX * time);
double futureY = e.getY() + (deltaY * time);
```

3.4.6. Girando el arma al punto previsto

Por último, debemos apuntar nuestro cañon al lugar previsto de impacto:

```
setTurnGunLeft(gunBearingTo(futureX, futureY));
```

4. Investigación y desarrollo propio

A partir de aquí el trabajo será individual de cada alumno (puede solaparse con el paso 3). Deberéis investigar/prever a vuestros adversarios (los conocidos, y los de vuestros compañeros). Aplicar las técnicas que consideréis más útiles para intentar quedar lo más arriba posible en la tabla de clasificación.

5. Fuentes de información

- Wikipedia
- GhatGPT
- Modelos de Inteligencia Artificial (Ed. Marcombo)
- https://iep.utm.edu/artificial-intelligence/
- Materiales MIA curso MEC-20230524