Robocodeの達人たちが明かす秘訣: 織込まれた壁の回避策

David McCoy
Writer
Independent

2002年 5月 01日

ロボットが壁に衝突しないようにしながらも、なおかつ、コーナーにはまってしまったり、行きたい方向から大きくそれてしまったりすることのないアルゴリズムを考案するのは難しいことです。簡単な方法は、 織込まれた壁の回避 です。このヒントでは、David McCoyが、この手軽な技法を実装する方法を紹介します。

このシリーズの他の記事を見る

「敵の動きを追跡する」で構築したロボットに少し追加するだけで、既存の、または問題の多い移動アルゴリズムに、織込まれた壁の回避を追加できます。織込まれた壁の回避では、希望する方向に、ロボットが壁にどれほど近いかに基づいた安全な方向を加味することにより、可能な限り最善の方向を見つけようとします。

共通の数学演算のためにヘルパー・メソッドを追加する

ロボットに追加する最初のものは、頻繁に利用される数学アルゴリズムのためのいくつかのヘルパー・メソッドです。

`calculateBearingToXYRadians()` メソッドは、 `java.lang.Math` の `atan2()` メソッドを使用して、`sourceX,sourceY` から `targetX,targetY` への絶対方位を計算した後、その値を `sourceHeading` に対する相対方位に変換します。

さらに、`normalizeAbsoluteAngleRadians()` メソッドと `normalizeRelativeAngleRadians()` メソッドも必要です。

リスト 1. Math helper methods

```
private static final double DOUBLE_PI = (Math.PI * 2);
private static final double HALF_PI = (Math.PI / 2);

public double calculateBearingToXYRadians(double sourceX, double sourceY,
    double sourceHeading, double targetX, double targetY) {
        return normalizeRelativeAngleRadians(
           Math.atan2((targetX - sourceX), (targetY - sourceY)) -
               sourceHeading);
    }

public double normalizeAbsoluteAngleRadians(double angle) {
   if (angle < 0) {
        return (DOUBLE_PI + (angle % DOUBLE_PI));
```

```
    } else {
        return (angle % DOUBLE_PI);
    }
}

public static double normalizeRelativeAngleRadians(double angle) {
    double trimmedAngle = (angle % DOUBLE_PI);
    if (trimmedAngle > Math.PI) {
        return -(Math.PI - (trimmedAngle % Math.PI));
    } else if (trimmedAngle < -Math.PI) {
        return (Math.PI + (trimmedAngle % Math.PI));
    } else {
        return trimmedAngle;
    }
}
```

AdvancedRobotを拡張して前後入れ替え機能を設ける

次に、`AdvancedRobot` クラスの機能を拡張して、逆方向に移動するための前後入れ替え操作を可能にするいくつかのヘルパー・メソッドを用意します。

- `getRelativeHeading()` メソッドは、ロボットの現在の向きに対する正しい方向を計算するオーバーヘッドを処理します。
- `reverseDirection()` メソッドは、まさにその通りのものです。 `direction` インスタンス変数を切り替えて、ロボットの向きを逆転させます。 減速には時間がかかるため、向きを逆転するまでに、ロボットは速度に応じて最大4フレームまで同じ向きに移動し続けることがあります。
- `setAhead()` および `setBack()` メソッドは、 `AdvancedRobot` クラスの同じ名前のメソッドをオーバーライドします。これらのメソッドは、現在の向きに対するロボットの相対的な速度を設定し、必要に応じて `direction` インスタンス変数を調整します。これは、相対的な操作が、いつも確実にロボットの現在の移動方向に対するものとなるようにするための処置です。
- `setTurnLeftRadiansOptimal()` および `setTurnRightRadiansOptimal()` メソッドは、`(Math.PI / 2)` より大きな回転に対してロボットの向きを逆転させます。これらのメソッドは、 `adjustHeadingForWalls` メソッドを使用するときに使いたくなります。このメソッドについては、後で説明します。

注: 私は、getterおよびsetterメソッドを使用するのではなく、 `direction` インスタンス変数に直接アクセスしています。これは、一般には望ましいプログラミング習慣ではありませんが、データ・アクセスを高速化するために、私のロボットのコードではしばしばそうしています。

リスト 2. Robot helper methods

```
public double getRelativeHeadingRadians() {
    double relativeHeading = getHeadingRadians();
    if (direction < 1) {
        relativeHeading =
                normalizeAbsoluteAngleRadians(relativeHeading + Math.PI);
    }
    return relativeHeading;
}

public void reverseDirection() {
    double distance = (getDistanceRemaining() * direction);
    direction *= -1;
    setAhead(distance);
```

```
}

public void setAhead(double distance) {
    double relativeDistance = (distance * direction);
    super.setAhead(relativeDistance);
    if (distance < 0) {
        direction *= -1;
    }
}

public void setBack(double distance) {
    double relativeDistance = (distance * direction);
    super.setBack(relativeDistance);
    if (distance > 0) {
        direction *= -1;
    }
}

public void setTurnLeftRadiansOptimal(double angle) {
    double turn = normalizeRelativeAngleRadians(angle);
    if (Math.abs(turn) > HALF_PI) {
        reverseDirection();
        if (turn < 0) {
            turn = (HALF_PI + (turn % HALF_PI));
        } else if (turn > 0) {
            turn = -(HALF_PI - (turn % HALF_PI));
        }
    }
    setTurnLeftRadians(turn);
}

public void setTurnRightRadiansOptimal(double angle) {
    double turn = normalizeRelativeAngleRadians(angle);
    if (Math.abs(turn) > HALF_PI) {
        reverseDirection();
        if (turn < 0) {
            turn = (HALF_PI + (turn % HALF_PI));
        } else if (turn > 0) {
            turn = -(HALF_PI - (turn % HALF_PI));
        }
    }
        setTurnRightRadians(turn);
}
```

織込まれた壁の回避を追加する

追加する必要のある最後のメソッドは、`adjustHeadingForWalls()` です。

このメソッドの前半では、ロボットが壁にどれほど近いかに基づいて、 安全な x,y位置を選びます (これは、ロボットの現在のxまたはy位置か、または、ロボットが壁に近い場合は競技場の中心点です)。メソッドの後半では、"安全な" 地点への方位を計算し、ロボットが壁にどれほど近いかに応じて、希望する方向にこの方位を加味します。

ロボットがどれほど壁のことを気にするかは、 `WALL_AVOID_INTERVAL` および `WALL_AVOID_FACTORS` 定数を使って調整できます。

リスト 3. The wall avoidance method

```
private static final double WALL_AVOID_INTERVAL = 10;
private static final double WALL_AVOID_FACTORS = 20;
private static final double WALL_AVOID_DISTANCE =
        (WALL_AVOID_INTERVAL * WALL_AVOID_FACTORS);
```

```
private double adjustHeadingForWalls(double heading) {
    double fieldHeight = getBattleFieldHeight();
    double fieldWidth = getBattleFieldWidth();
    double centerX = (fieldWidth / 2);
    double centerY = (fieldHeight / 2);
    double currentHeading = getRelativeHeadingRadians();
    double x = getX();
    double y = getY();
    boolean nearWall = false;
    double desiredX;
    double desiredY;

    // If we are too close to a wall, calculate a course toward
    // the center of the battlefield.
    if ((y < WALL_AVOID_DISTANCE) ||
            ((fieldHeight - y) < WALL_AVOID_DISTANCE)) {
        desiredY = centerY;
        nearWall = true;
    } else {
        desiredY = y;
    }
    if ((x < WALL_AVOID_DISTANCE) ||
             ((fieldWidth - x) < WALL_AVOID_DISTANCE)) {
        desiredX = centerX;
        nearWall = true;
    } else {
        desiredX = x;
    }

    // Determine the safe heading and factor it in with the desired
    // heading if the bot is near a wall
    if (nearWall) {
        double desiredBearing =
           calculateBearingToXYRadians(x,
                                       y,
                                       currentHeading,
                                       desiredX,
                                       desiredY);
        double distanceToWall = Math.min(
                Math.min(x, (fieldWidth - x)),
                Math.min(y, (fieldHeight - y)));
        int wallFactor =
                (int)Math.min((distanceToWall / WALL_AVOID_INTERVAL),
                              WALL_AVOID_FACTORS);
        return ((((WALL_AVOID_FACTORS - wallFactor) * desiredBearing) +
                 (wallFactor * heading)) / WALL_AVOID_FACTORS);
    } else {
        return heading;
    }
}
```

すべてをまとめる

残りの作業は簡単です。現在の移動アルゴリズムをそのまま利用し、 `adjustHeadingForWalls()` メソッドの結果を提供することにより、壁を回避します。

物事を簡単にするために、サンプル・ロボット (この技法に追加する必要のあるソース・コードについては、 参考文献を参照) は、方向の変化として0を要求することにより、いつも直線上を移動しようとします。

リスト 4. Wall avoidance method

```
public void run() {
    while(true) {
        setTurnRightRadiansOptimal(adjustHeadingForWalls(0));
        setAhead(100);
        execute();
    }
}
```

この技法は、ただこれだけのことです。単純ですが、効果的です。

著者について

David McCoy

David McCoyは、8か月前に同僚からRobocodeを紹介されて以来、すぐにそのとりこになりました。彼の競技用ロボットdroid.PrairieWolfは、Gladiatorial Leagueの混戦を呈する競技会で首位にランクされたことがあります。Davidの連絡先は、dsmccoy@velocitus.net です。