Robocode の達人たちが明かす秘訣: 反重力移動

バトルフィールド上の特定の地点を避け、移動パターンを作成し、敵の弾丸をかわす

Alisdair Owens
Student
Student

2002年 5月 01日

反重力移動 (多くの修正版がある) は、大部分の熟練した Robocode プログラマーが採用する移動のタイプです。この移動方法では、マップ上で避けるべき地点を定義することができ、移動パターンを容易に作成し、敵の弾丸をかわすことができます。Alisdair Owens は、この有用な技法を実装する方法を紹介し、試運転のためのサンプル・ロボットを提供します。

反重力移動は、パターン分析型のロボットを欺くのに役立つ高度に柔軟な技法であり、バトルフィールド上で避けるべき特定の地点 (重力ポイント という) を定義することができます。重力ポイントにはそれぞれ、強さ が割り当てられます。この強さの x 方向と y 方向の成分を分析することにより、すべての敵ロボットを効果的に回避できます。(このヒントで使われる用語を理解する助けとして、「反重力の用語」という補足記事をご覧ください。)

この記事の前半では、基本的な反重力の技法を紹介し、後半では、この基本的なコードの根本的な限界を回避するためのアイデアについて説明します。

反重力の用語

重力ポイント: Robocode バトルフィールド上に定義するポイントで、そこから反発力または誘引力が働くようにしたい領域のことです。

力: 各重力ポイントには、力 もしくは 強さ を割り当てます。この値が大きいほど、その力を発生させているポイントに対してロボットが誘引または反発される度合いが大きくなります。

力の成分: それぞれの力には、x (水平) 方向に働く成分 と、y (垂直) 方向に働く成分があります。角度が 45 度の場合は、x 方向と y 方向の成分が等しくなります。角度が 90 度の場合は、x 方向にのみ力が働き、角度が 0 度の場合は、y 方向にのみ力が働きます。

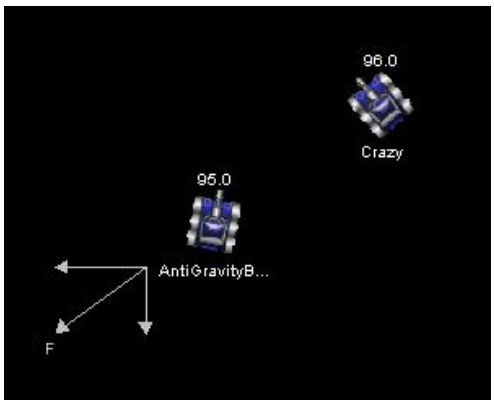
力を分析する: 複数の力が相互に作用するときに、生成される合計の力を算出するプロセスのことです。たとえば、x 方向に -200 の力が働いているときに、同じく x 方向に 300 の力も働いているとすると、生成される全体の力は x 方向に 100 となります。

反重力移動の背後にある数学

三角法の実用的な知識を持っていれば、反重力の背後にある数学はごく簡単に理解できます。

図 1 で、"F" という矢印は、Crazy から AntiGravityBot に対する力の方向を示しています。力は、x 方向と y 方向の成分として考えるとことができます。これは、図にある他の 2 つの矢印で示されています。力を分析すると、すべての重力ポイントからの力を x 方向と y 方向で単純に加算することができ、x 方向と y 方向における合計の力を算出できます。

図 1. 力を分析する



遠く離れたロボットから自分のロボットへの影響を小さくするには、重力ポイントから自分のロボットに対する力の計算に、`force = strength/Math.pow(distance,2)` という関数を使用しなければなりません。この関数で、strength は重力ポイントのパワー、distance は重力ポイントと自分のロボットの距離です。2 というべき乗値は、固定ではありません。重力ポイントに非常に近づいたときにのみそのポイントを避けるようにするには、3 という値を使用することもできます。

コード

この後のリストは、基本的な反重力システムのコードを示しています。リスト 1 は、メインの反重力関数です。この関数は、ベクトル内のすべての重力ポイントをたどり、力を解分析し、ロボットを正しい方向へ移動させます。敵ロボットを反発ポイントにすることをお勧めします。そのためには、バトルフィールドについてかなり最新の全体像を保持しなければならず、レーダーをかなり頻繁に回転させる必要があります。

リスト 1. The anti-gravity workhorse: antiGravMove()

```
void antiGravMove() {
    double xforce = 0;
    double yforce = 0;
    double force;
    double ang;
    GravPoint p;

    for(int i = 0;i<gravpoints.size();i++) {
        p = (GravPoint)gravpoints.elementAt(i);
        //Calculate the total force from this point on us
        force = p.power/Math.pow(getRange(getX(),getY(),p.x,p.y),2);
        //Find the bearing from the point to us
```

```
        ang =
    normaliseBearing(Math.PI/2 - Math.atan2(getY() - p.y, getX() - p.x));
        //Add the components of this force to the total force in their
        //respective directions
        xforce += Math.sin(ang) * force;
        yforce += Math.cos(ang) * force;
    }

    /**The following four lines add wall avoidance.  They will only
    affect us if the bot is close to the walls due to the
    force from the walls decreasing at a power 3.**/
    xforce += 5000/Math.pow(getRange(getX(),
      getY(), getBattleFieldWidth(), getY()), 3);
    xforce -= 5000/Math.pow(getRange(getX(),
      getY(), 0, getY()), 3);
    yforce += 5000/Math.pow(getRange(getX(),
      getY(), getX(), getBattleFieldHeight()), 3);
    yforce -= 5000/Math.pow(getRange(getX(),
      getY(), getX(), 0), 3);

    //Move in the direction of our resolved force.
    goTo(getX()-xforce,getY()-yforce);
}
```

リスト 2 に示したヘルパー・メソッドを利用すると、自分のロボットと敵ロボットの距離を保つために最も効率の良い位置へ移動できます。

リスト 2. Helper methods

```
/**Move in the direction of an x and y coordinate**/
void goTo(double x, double y) {
    double dist = 20;
    double angle = Math.toDegrees(absbearing(getX(),getY(),x,y));
    double r = turnTo(angle);
    setAhead(dist * r);
}

/**Turns the shortest angle possible to come to a heading, then returns
the direction the bot needs to move in.**/
int turnTo(double angle) {
    double ang;
    int dir;
    ang = normalisebearing(getHeading() - angle);
    if (ang > 90) {
        ang -= 180;
        dir = -1;
    }
    else if (ang < -90) {
        ang += 180;
        dir = -1;
    }
    else {
        dir = 1;
    }
    setTurnLeft(ang);
    return dir;
}

/**/Returns the distance between two points**/
double getRange(double x1,double y1, double x2,double y2) {
    double x = x2-x1;
    double y = y2-y1;
    double range = Math.sqrt(x*x + y*y);
    return range;
}
```

最後に、リスト 3 の `GravPoint` クラスは、重力ポイントについて必要なすべてのデータを保持します。反発させるためには、`power` を負の値にしなければならないことに注意してください。

リスト 3.GravPoint class

```
class GravPoint {
    public double x,y,power;
    public GravPoint(double pX,double pY,double pPower) {
        x = pX;
        y = pY;
        power = pPower;
    }
}
```

このヒントのソース全体は、参考文献からダウンロードできます。

振る舞いを改善する

リスト 1～3 のコードは、理にかなった振る舞いをロボットにさせますが、戦闘におけるパフォーマンスはあまり良くありません。ロボットは全般的に他のロボットから離れた位置を保ちますが、壁ぎわで立ち往生してしまう傾向があります。その理由は、たとえばロボットが下側の壁に達すると、自分より下にロボットは 1 台もないからです。したがって、ロボットを壁から押し戻す力は、壁そのものによる反発力しかありません。壁の反発力には限られた範囲しかないため、貧弱な振る舞いになってしまうわけです。

この問題と闘うために、私は、競技場中の一連のポイントにおける力を合計するシステムを使用しています。その後、力の総合計の平均値より力が大きいポイント (つまり、近くにロボットがいるポイント) に反発力の値を割り当て、平均値より力が小さいポイントに誘引力の値を割り当てます。その上で、あらためて、これらの新しいポイントから自分のロボットに対する力を分析します。誘引ポイントを割り当てる際には、きわめて慎重にしなければなりません。自分のロボットが誘引ポイントに近づくと、ロボットはその周囲をうろつくだけで、決して離れなくなってしまうからです。その理由から、これらの中間ポイントの位置をランダムに割り当てることと、その位置を定期的に変更することをお勧めします。

この拡張のためのコードを作成することは、読者にお任せします。基本的な原理はまったく同じままで、上記のコードに少しの変更を加えるだけですので、安心してください。もう少しのヒントとして、中間ポイントから自分のロボットへの力を計算する際に、`force = strength/Math.pow(distance,1.5)` を使うことをお勧めします。

その他の拡張点

反重力は、非常に柔軟性のある技法です。そのため、実現できる振る舞いすべてを説明することは実際的でありません。しかし、興味深い拡張点のいくつかを、ここで紹介しておきます。

標的の選択: 攻撃しやすい標的、または弱っている標的の反発値を低く設定すれば、それらの標的の近くに移動して、弱い相手を仕留めることができます。

ランダム化: かなり定期的に、x および y 方向の力からランダムな値を加算または減算することにより、いっそうランダムな動きを実現できます。時には停止することさえ可能でしょう。そうす

れば、敵の標的合わせシステムを欺くことができます。この振る舞いを実装することを、強くお勧めします。

混戦で弾丸をかわす: 敵から自分に向けて弾丸が発射された時点を検知できるのであれば、発射された弾丸を反重力ポイントとしてモデル化できます。たとえば、敵の弾丸が直線方式の標的合わせで発射されたと想定される場合は、その重力ポイントの位置を各動作 (turn) ごとに更新することにより、弾丸をかわすことができます。しかし、この拡張を完全に実現したロボットはまだありません。

リーダーに従う: この拡張には、自分のロボットに誘引ポイントを設定して従わせることが関係しています。こうして、そのポイントを移動することにより、自分の望むパターンを (Robocode の物理法則の範囲内で) 生み出すことができます。しかも、標準的な反重力の壁反発には従います。

ダウンロード

内容	ファイル名	サイズ
Source code	j-antigrav.zip	12KB

著者について

Alisdair Owens

Alisdair Owens は、英国サウサンプトン大学のコンピューター・サイエンスの学生です。Java 言語によるプログラミングに2年間携わっており、Robocode に特に関心を寄せています。