世界模型的层次分析 V1.0

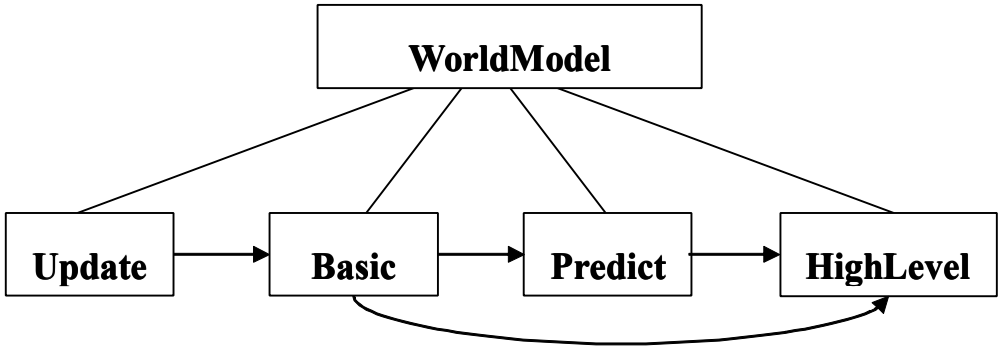
施国强 2007.5.31

世界模型（WorldModel）是 Agent 用来描述周围环境信息的。Agent 应该能够知道球场上自己以及其他球员和球的速度和方向。显然，Agent 应该具有一个模块用于保存球场上各对象（22 个球员、球以及球场的一些标志）的信息，所以，World Model 是必须的。

一、层次和关系

参照 UvA2D 的底层，我们把 WorldModel 划分成 4 个部分：Update、Basic、Predict、HighLevel。

相互关系如下图：



Update: 消息解析模块在对服务器发送来的消息进行解析后，更新存放在 Basic 中的信息，并计算球员及球的位置和速度。

Basic: 主要用于信息的存贮。这些信息将会被 WorldModel 中的 Predict、HighLevel 层以及其他类如 skill 层和 Decision 使用。

Predict: 在 Basic 层的基础上，做关于 Agent 自身、其他球员和球状态的预测。

HighLevel: 在前两者基础之上的一些基本函数，如判断 Agent 自身是否离球最近、是否在某个区域内，截球点的计算等。

以上只是简单的介绍，接下来会较详细的分析各个部分。

二、Basic 部分

前面已经说过，这个部分主要用于信息存储。

1、关于初始化时的各种参数（参考 0.5.4 版 server 的 rcssserver3D.rb）

- FieldLength： 球场的长度（随机数：100.0~110.9）
- FieldWidth： 球场的宽度（随机数：64.0~75.9）
- FieldHeight： 球场的高度（定值：40）
- GoalWidth： 球门的宽度（定值：7.32）
- GoalDepth： 球门的深度（定值：2.0）
- GoalHeight： 球门的高度（定值：1.6）
- BorderSize： 边界宽度（定值：10）

- AgentMass：　球员质量（定值：75.0）
- AgentRadius：　球员半径（定值：0.22）
- AgentMaxSpeed：　球员的最大速度（定值：10.0）
- BallRadius：　球的半径（定值：0.111）
- BallMass：　球的质量（随机数：0.41~0.45）
- unum：　球员的号码（1-11）
- side：　球队是在左边还是在右边

以上数据都会在初始化时赋以定值，比赛开始后不会再发生变化。

2、信息分为以下几类：

a) 环境信息：包括球场的长宽高，球门的宽度、深度和高度，边界宽度，以及我方是在左边还是在右边。即：FieldLength、FieldWidth、FieldHeight、GoalWidth、GoalDepth、GoalHeight、BorderSize、side。

b) 对象的静态信息：包括 Agent 的质量、半径、最大速度和球员号，球的半径和质量。即：AgentMass、AgentRadius、AgentMaxSpeed、unum 、BallRadius、BallMass。

c) 对象的动态信息：包括 Agent 的位置、速度，视角中的平面角（Pan）和高度角（Tilt），描述体力状态的温度和电量。以及其他球员以及球的位置和速度。其中关于位置和速度的数据是不可以直接从消息中获得的，在 Update 部分有相关的算法负责计算。

d) 比赛状态的信息：包括比赛的时间周期（0~60000，每个周期 10 毫秒）、双方的队名、比赛模式、比分等。

已知的比赛模式如下：

BeforeKickOff：开球前
KickOff_Left：左方球队开球
KickOff_Right：右方球队开球
PlayOn：比赛中
KickIn_Left：左方球队边界球
KickIn_Right：右方球队边界球
corner_kick_left：左方球队角球
corner_kick_right：右方球队角球
goal_kick_left：左方球队球门球
goal_kick_right：右方球队球门球
offside_left：左方球队越位
offside_right：右方球队越位
GameOver：比赛结束
Goal_Left：左方球队进球
Goal_Right：右方球队进球
free_kick_left：左方球队任意球
free_kick_right：右方球队任意球
unknown：未知模式

3、历史信息的存贮

用于存储感知到的对象的位置及对应的时间周期。这里的对象包括 Agent 自身、其他球员和球。每次程序接收到 server 发送来的视觉信息后，从中解析出所含对象的信息（即相对于 Agent 的极坐标），（这些对象往往不包含球场上所有的队员和球场标志，甚至不包含球，也由此突显出预测算法的重要性），在经过相关的计算后得出对象的位置（笛卡尔坐标），并连同时间周期一起保存到该对象对应的信息存储队列中。在程序的许多地方都用到了队列，为减少代码的重复，在此可以使用队列模板。并且所存贮的历史信息也要考虑到时效性，因此针对一个对象没有必要存储过多的历史信息，只要够用就行，一般 10 个就应该可以了。队列模板可以采用循环队列的方式。

历史信息主要用于计算和预测各个对象的位置和速度。

三、Update 部分

比赛在进行的过程中，球场上的球员和球的状态不断的发生变化。然而 Agent 必须针对球场上的状况做出反映。要求世界模型的更新部分必须较准确较时效的对世界模型中的各个对象的状态进行更新，在有必要的情况下还要使用预测算法。关于预测算法将在下一节介绍。本节介绍针对接收到的视觉信息及历史信息计算 Agent 和球的位置和速度。

1、Agent 自定位算法

Agent 每隔 20 个周期能接收到 1 个视觉消息。收到的时间通报是不含视觉消息的。然而由于 Agent 的视角和所处位置的影响，Agent 不能看到球场上所有的球员和球场标志，甚至看不到球，反映在所接收到的视觉消息上就是，视觉消息中不含这些对象的相对位置信息。所有的极坐标都是这些对象相对于 Agent 的相对极坐标。可以利用 Agent 与球场标志的相对极坐标计算出 Agent 的位置。进而可以算出其它可见对象的位置。使用适当的算法后可以获得 Agent、球和其他球员的速度。可以说 Agent 自定位是世界模型维护的基础。

下面是一个视觉消息：

S561 571 (Vision (Flag (id 1_r) (pol 83.13 -30.82 -0.23)) (Flag (id 2_r) (pol 76.95 21.70 -0.19)) (Goal (id 1_r) (pol 72.36 -8.33 -0.05)) (Goal (id 2_r) (pol 71.60 -2.63 -0.14)) (Ball (pol 19.33 -21.13 -0.21)) (Player (team AIAI_3D_2007) (id 3) (pol 8.50 -86.86 -0.15)) (Player (team AIAI_3D_2007) (id 4) (pol 5.52 -84.89 -0.01)) (Player (team AIAI_3D_2007) (id 5) (pol 14.02 -89.83 0.16)) (Player (team AIAI_3D_2007) (id 6) (pol 17.83 -51.67 0.16)) (Player (team AIAI_3D_2007) (id 7) (pol 11.00 0.03 0.05)) (Player (team AIAI_3D_2007) (id 8) (pol 10.63 -41.15 0.06)) (Player (team AIAI_3D_2007) (id 9) (pol 18.37 -22.54 -0.11)) (Player (team AIAI_3D_2007) (id 10) (pol 16.75 10.09 -0.17)) (Player (team AIAI_3D_2007) (id 11) (pol 23.72 -45.68 0.16)) (Player (team WrightEagle2005) (id 1) (pol 20.28 -19.99 0.21)) (Player (team WrightEagle2005) (id 2) (pol 18.95 9.17 0.06)) (Player (team WrightEagle2005) (id 3) (pol 25.27 -42.31 -0.01)) (Player (team WrightEagle2005) (id 4) (pol 33.03 -1.69 0.04)) (Player (team WrightEagle2005) (id 5) (pol 34.14 -14.91 -0.01)) (Player (team WrightEagle2005) (id 6) (pol 26.04 3.88 -0.09)) (Player (team WrightEagle2005) (id 11) (pol 68.98 -5.57 0.22)) (Player (team WrightEagle2005) (id 7) (pol 30.44 -31.37 -0.01)) (Player (team WrightEagle2005) (id 8) (pol 42.84 -9.26 0.14)) (Player (team WrightEagle2005) (id 9) (pol 43.31 -5.83 0.19)) (Player (team WrightEagle2005) (id 10) (pol 46.22 -21.66 -0.06)))(GameState (time 0.00) (playmode

BeforeKickOff))

这个消息中可以用于自定位的相对极坐标只有 4 个。下面介绍一种简易的自定位算法：

由于每个球场标志的绝对坐标都是已知的，可以分别利用这 4 个标志计算出 Agent 的绝对坐标，再将计算出的 4 个绝对坐标累加求平均。Agent 相对于球场标志的相对极坐标是存放在一个 1 行 8 列的数组中的。实际的算法中如何区分这 8 个相对极坐标是否可用呢？程序的消息解析模块在对消息进行解析并更新 WorldModel 时，写入相对极坐标的同时也记录下它的有效时间周期。因此可以通过判断相对极坐标对应的时间周期是否与当前的视觉消息的时间周期相等，来决定是否可以用于自定位算法。

以下是算法的大致表示：

```
Vector3f CalculateAgentPos()
{
    int i;
    int j=0;
    Vector3f vPos;
    Vector3f vTotalPos(0.0,0.0,0.0);
    for(i=0; i<8; i++)
    {
        if (RelFlagPolarPos[i].Cycle != CurrentVisionMsgCycle) continue;
        j++;
        vPos= FlagVectorPos[i] - RelFlagPolarPos[i]. PolarPos.PolarToVector3f();
        vTotalPos += vPos;
    }
    return vTotalPos/j;
}
```

这种算法只是提供一个思路，实际的误差较大，不能用于球队程序中。在 AIAI3D 底层中采用的是卡尔曼滤波算法，实际效果较好。有条件的球队可以同时采用粒子滤波和卡尔曼滤波相结合的方法，进一步减小误差。

2、计算其他对象的位置

在 Agent 自定位的基础上，就可以计算其他对象的位置，因为 Agent 相对于其他对象的相对位置是可知的。可以利用以下公式进行计算：

$$\vec{P}_{obj} = \vec{P}' + \vec{P}_{self}$$

其中 $\vec{P}_{obj}$ 为物体的绝对坐标，$\vec{P}_{self}$ 为 Agent 自身的坐标，$\vec{P}'$ 为由相对极坐标转化后的直角坐标。关于极坐标的转化在几何模型中有说明。

3、计算 Agent 的速度

通过 Agent 自定位算法已经可以计算出 Agent 的位置。然而，Agent 的速度是无法从 server 发送来的消息中获取的。这时可以利用 v=s/t 来计算。具体实现

就是令位移等于两次的位置相减，因为位置是用三维笛卡尔坐标定义的，所以两次的位置之差是一个三维向量，时间差就是两次位置对应的时间之差。这里使用到了两个位置及其对应的时间周期，实际上已经使用到了历史信息。然而，Agent 自定位算法是有误差的，仅仅使用两个视觉信息，必然会有较大的误差。这时可以使用多个历史视觉信息来减小误差。以下是我们使用的方法：
现在要对 Agent 的视觉历史信息队列进行访问，设 P_n 是队列中倒数第 n 个信息中的 Agent 的位置，P_1 即表示最近的一个位置，相应的 T_n 表示 P_n 所对应的时间周期。
const n=5;
const dMod1=2;
const dMod2=0.2;
Vector3f vStage12= P_1 - P_2;
double dTime12=(T_1 - T_2)*0.01;
Vector3f vVelocity12= vStage12/ dTime12;
Vector3f vStage2n= P_2 – P_n;
double dTime2n=(T_2 – T_n)*0.01;
Vector3f vVelocity2n= vStage2n/ dTime2n;
Vector3f vDeltaVel= vVelocity12 - vVelocity2n;
double dDeltaVelMod= vDeltaVel.getMod(); //获取向量的模长
if (dDeltaVelMod> dMod1) return Agent 原来的速度;
else if (dDeltaVelMod> dMod2) return vVelocity12;
else return (vVelocity12 + vVelocity2n)/2;
算法分析：这个算法计算了两个时间段的速度：V_{12} 和 V_{2n}，这里的速度是三维向量，作这两个速度的差，它也是一个三维向量，计算这个向量的模长，如果这个模长较长，如大于 2，那么可能是因为越位之类的情况，server 自动将球员放置到某个位置，导致在这个时间段的速度较大，然而这个速度是没有用的，在这种情况下我们只有使用球员原来的速度。当这个速度之差的模长不是很大时，如大于 0.2，可能是在 T_2 时刻球员的运动状态发生了较大的变化，如受到较大的撞击或受到 Drive 的作用，这时我们使用 T_1 和 T_2 时刻计算出的速度 V_{12}，由于运动状态的较大变化，V_{2n} 也是不可取的。当两个速度之差的模长很小时，为了减小误差，我们可以使用两个速度的平均值。算法中的几个常量可以视情况做调整。算法最终得到的是 T_1 时刻的大致速度。

4、计算其他球员和球的速度

算法和计算 Agent 本身速度的算法大致相同。不同之处是：关于 Agent 的视觉历史信息队列中存放的信息之间的时间间隔是基本相同的（20 个周期），除非遇到特殊情况，如 server 发送的信息丢失。然而由于视角和 Agent 位置的影响，Agent 不能看到所有的对象，server 发送的视觉信息里也就不会包含所有对象相对于 Agent 的极坐标。那么这些对象对应的历史信息队列中相邻两项的时间周期之差往往是无法预料的。

在实际的程序中，可以通过改变 Agent 的 Pan 和 Tilt 这两个角度来改变 Agent 的视角。由于 Agent 能不能看见球是比较重要的，所以程序在经过一些算法后，可以尽量保证 server 发送的视觉信息里包含有球的信息。这也就使得计算球的速度可以使用和计算 Agent 速度完全相同的算法。

对于队友和对方球员，由于可能有较长的周期内获取不到它们的信息，他们的速度也只能使用上一次获得它们信息时计算的速度来代替，并且上次获得它们信息的时间如果离当前的时间周期越远，这个速度的可信度越差，似乎没有什么较好的方法了。

四、Predict 部分

前面已经多次提到，server 发送给 Agent 的视觉消息里往往不包含所有球员的信息，甚至不包含球。然而，在 Agent 的高层决策部分，需要把握球场上的状况，要知道其他球员和球的状态。如果当前 Agent 并不能发现某些需要了解的对象，那只能利用以前获取的信息对当前的状态进行预测了。

我们再来看一下几个视觉消息的前一段：

S241 251 (Vision …)
S261 271 (Vision …)
S281 291 (Vision …)
S301 311 (Vision …)
S321 331 (Vision …)
S341 351 (Vision …)

这是我从 Agent 获取到的所有消息中截取的连续的一部分，具体的视觉信息内容在此省略。S 后有两个整形数字，第一个数字表示 server 发送视觉消息的时间，也就是说这一条视觉消息反映的就是这个时间周期的状况。S 后的第二个数字表示 Agent 接收到这条消息的时间周期。从以上可以明确的分析出：server 每隔 20 个周期发送一条视觉消息，直到 Agent 接收到这条消息之间有 10 个周期的延迟。这个延迟是由于传输导致的。那么 Agent 发送给 server 的消息也有 10 个周期的延迟。如果我针对第 241 个周期的状况做出决策，等 Agent 把动作命令发送到 server，已经是第 261 个周期了（不计算程序算法的耗时），球场的状况发生了变化，Agent 的动作命令也可能不适合于新的状况了。为了缓解这个问题，我们仍然要使用预测算法。比如说，针对第 241 个周期的视觉信息，在 Update 部分对 WorldModel 进行更新之后，再使用预测算法预测第 261 个周期的球场上的状况，在此基础之上再进行高层决策。这样可以尽可能的保证动作命令的有效性。

基于预测算法做高层决策，要求预测算法能较准确的预测出未来的状态，如果效果较差，那还不如不用预测算法，直接在 Update 的基础上做决策。AIAI3D 在这一方面，由于各方面的限制，并没有使用训练器收集数据然后再推导公式，而是直接参照了刘汝佳的《RoboCup3D 仿真组中世界模型的维护》。在这篇文章中，通过收集大量的数据和推导，给出了计算 Agent 和球的位置和速度的公式，可以直接使用这些公式预测出 Agent 和球在未来的状态。至于预测其他球员，那就要自行写算法了。

1、预测 Agent 的状态

使用如下的公式：

$$v(t) = v_m + e^{-st}(v_0 - v_m)$$

$$S(t) = \frac{v_0 - v_m + v_m st - e^{-st}(v_0 - v_m)}{s}$$

或 $S(t) = (v_0 - v(t))/s + v_m * t$

其中 v_m 是 Agent 受到的驱动力的大小， v_0 是 Agent 的初速度，s=2.51038。假设 t_0 时刻 Agent 的速度为 v_0，位置是 s0，则 t_1 时刻的速度为 $v(t_1 - t_0)$，位置是 s0+ $S(t_1 - t_0)$。因为球员不可以跳起，所以我们只考虑 Agent 在水平面上的速度和位置。无论在 X 轴方向还是 Y 轴方向，球员的速度和受到的驱动力成正比，球员的最大速度是 1.616m/s，对应的驱动力为 100，也就是说驱动力为 50 时，球员的速度为 0.808m/s。

2、预测球的状态
使用如下的公式：

$$v(t) = v_0 e^{-st}$$

$$S(t) = \frac{v_0(1 - e^{-st})}{s}$$

其中 v_0 是球的初速度，s=0.2148156/BallMass，球的质量是不定的，这里的参数s也不是定值。假设 t_0 时刻球的速度为 v_0，位置是 s0，则 t_1 时刻的速度为 $v(t_1 - t_0)$，位置是 s0+ $S(t_1 - t_0)$。注意：球是可以在空中的，也就是要考虑 Z 轴的运动，以上两个公式只使用于计算球着地时在 X 轴方向和 Y 轴方向上的速度和位移，对于 Z 轴方向需要建立空中模型，比较复杂。

3、预测队员的状态

如果关于某个队员的视觉信息所对应的时间周期距离当期的周期较远，那么这个视觉信息就没有使用价值了

如果在上一次使用视觉信息计算出（并非预测）某个队员的位置和速度之后，过了很多个时间周期后才收到新的视觉信息，那么先前计算的位置和速度往往很难反映最近的状况，是不可取的，在这种情况下不妨认为这个球员速度没有发生变化，对于队友可以认为它回到了自己的本位点，对于对方球员可以用 s=vt 计算出位移，进而计算出位置。虽然误差很大，但似乎没有什么较好的方法了。

如果当前的周期距离上次收到某队员相关视觉信息的周期不远，比如相差 60 个时间周期，可以使用上一段中计算对方球员的方法来计算。

RoboCup3D 仿真是一种分布式多智能体系统。在这种系统中，对于对方球员的状态是难以预测的，因为我们无法得知对方的阵型和决策，误差必然很大，但是又必须做这个工作。对于队友的预测相对要好一点，因为我方队员的阵型和

决策是已知的，如果算法做的好，也可以使误差相对较小。上面说到的预测队员的状态的方法，比较简单，效果也不是太好，但对实际的高层决策也是有很大帮助的。

五、HighLevel 部分

高层决策是 Agent 针对球场上的状况做出反应。而这里的 HighLevel 并非是高层决策。它是对 WorldModel 中所有的数据和信息进行有效的组织，并为高层决策提供一些基本的函数调用。例如，判断 Agent 是否在某一块区域之内，寻找距 Agent 最近的我方和对方球员，计算球员跑到某一点需要的最短时间等等。