国際政治と情報（2005 年後期）

担当　田中明彦 tanaka@ioc.u-tokyo.ac.jp
TA　阪本拓人 takutos@nifty.com
保城広至 hoshiro@ioc.u-tokyo.ac.jp

第９回　複雑な行動をするエージェント（12 月 9 日）

●概略

○ 前回の宿題の解答例と解説
○ 記憶と行動：GetHistory 関数再び
○ エージェントの行動のモジュール化：ユーザー定義関数
○ 配列とデータの入出力
○ 宿題

●前回の宿題の解答例と解説

[５] 文化属性のカテゴリーの数をパラメータ化しコントロールパネルで操作できるようにした上で、カテゴリー数を 3 から 5 に増やして、[４]と同じような繰り返し試行を行ってみてください。ルールに加える変更は、カテゴリー数の数値を Universe に加えた変数で置き換えることと、配色の際の文字列から整数への変換式を書き換えることです。後者は若干の工夫を要します。

＞Universe にコントロールパネルで操作するための整数型変数（Num_Categories）を加え、ルール中に記述されている属性のカテゴリー数の数値 3 をこの変数で置き換えていきます。配色の部分のルールのみ以下に示します。ありうる文化の総数が変化しうるのでその部分を赤字のように可変化します。

```
My.Color = ( Color_White / 10 ^ Universe.Num_Categories ) * CInt( My.Culture )
```

[６] 文化変容モデルには、これ以上文化属性の分布が変化しなくなる定常状態が存在します。つまり、全ての Village エージェントについて、周囲の全てのエージェントの属性が全く同質か、または逆に全く異質な状態が成立しているならば、文化の変容は事実上起きなくなります。今回作ったモデルではシミュレーションの終了ステップを実行環境設定で外生的に決めていましたが、この論理をもとに、システムが定常状態に達したらシミュレーションが勝手に終了するようにモデルに変更を加えてみましょう。ルールの中でシミュレーションを終了させる関数として ExitSimulation( )（引数なし）という組み込み関数が存在するので、これを利用します。

＞詳しい解説は略しますが、以下の解答例では、Universe にシミュレーションを継続するか(1)否か(0)を制御する変数 Continue を追加した上で、Agt_Step の末尾にまず次のようなルールを書き加えて、各エージェントの文化属性がこれ以上変化しうる状態にあるかないかをチェックするという処理を行っています。

```
// 継続条件が満たされているか village ごとに判定
For Each Neighbor In Neighborhood

        // カテゴリーの一致数をカウント
        Num = 0
        For i = 1 To Universe.Num_Categories
                If Mid( My.Culture, i, 1 ) = = Mid( Neighbor.Culture, i , 1 ) Then
                        Num = Num + 1
                End If
        Next i

        // 完全に一致か完全に異質ではないなら継続
        If Num <> 0 AND Num <> Universe.Num_Categories Then
                Universe.Continue = 1
                break
        End If

Next Neighbor
```

全てのエージェントに対してこうしたチェックをした上で、実際にシミュレーションを終えるかどうかは Univ_Step_End で下記のように決定することになります。

```
// 継続条件が満たされていないならシミュレーション終了
If Universe.Continue == 0 Then
        ExitSimulation( )
End If
```

●本日のモデルとそのポイント

ルールエディタにカチャカチャ打ち込みつつ、マルチエージェント・シミュレーションのさまざまな技法を学ぶ実習形式の講義も、今日で一段落します。そこで最後に、いかにも社会科学らしいモデルをひとつ作って、講義を締めくくりましょう。それは、社会的なジレンマ状況としてよく知られた「囚人のジレンマ」（Prisoner's Dilemma, PD）のゲームを、コンピュータの中の多数のエージェントに繰り返しプレイさせるモデルです。

まず、簡単に囚人のジレンマについて説明しておきましょう。

複数行為者間の相互依存的な意思決定を数理的に形式化し分析する研究分野にゲーム理論があります。囚人のジレンマはこの研究分野で「発見」された、深刻なジレンマをはらむ意思決定状況であり、多くの場合、次のような挿話とともに語られます。

まず、二人の囚人（厳密に言うと容疑者）がいて、取調べを受けている状況を考えてください。この二人は共犯者で、コミュニケーションが遮断された二つの部屋で別々に取調べを受け、自白を迫られています。

このとき、囚人には「自白する」「黙秘する」という二つの選択肢があります。もし、お互いに黙秘すると、二人はともに懲役 1 年になります、しかし、自分が自白して、相手が黙秘すると、自分だけ釈放され、相手は懲役 10 年になります。反対に、相手が自白してしまって、自分が黙秘すると、相手が釈放、自分が懲役 10 年になってしまいます。また、お互いに自白すると、ともに懲役 5 年になります。

この意思決定状況を、刑期を負の利得と見なして、ゲーム理論の利得行列の形式にまとめると、次のようになります。左の数字が囚人１の利得（刑期)、右が囚人２の利得です。

		囚人 2	
		黙秘	自白
囚人 1	黙秘	1 年, 1 年	10 年,0 年
	自白	0 年, 10 年	5 年,5 年

さて、囚人たちにとって、最適な行動とはどのような行動でしょうか。お互いにコミュニケーションが取れず、拘束力のある申し合わせができない状況で、二人の囚人にとって最も合理的な行動は、相手を裏切って自白してしまうことです。実際、上の行列を見て分かるように、囚人１にとっては、囚人２が黙秘しようが自白しようが、自分が自白してしまう方がより少ない刑期で済みます。同じ推論は囚人２についても成り立ちます。つまり、個々人が合理的に振舞うことで、双方とも自白してしまう状態が帰結するわけです。

これが「ジレンマ」なのは、この（自白、自白）という結果が、双方が「協調」して黙秘する状態よりも、両者にとって明らかに望ましくないからです。しかし、相手の黙秘を信じて黙秘するという行動は、囚人個々人にとっては合理的ではありません。一方の黙秘

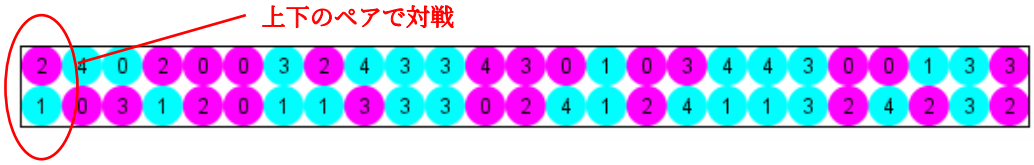
に付け込むことで、他方は自分だけ釈放を達成することができるからです。つまり、囚人のジレンマには、「社会的な望ましさ」と「個人的な合理性」が鋭く乖離している状況がきわめてシンプルな形で集約されていることになります。

このようなジレンマ状況が、二人の囚人の意思決定という特定の文脈を離れて、広範な社会現象に適用しうることは、容易に想像できるでしょう。たとえば、国際関係の文脈では、軍拡競争や市場開放をめぐる駆け引きがしばしば囚人のジレンマにたとえられてきました。ここでは詳しく触れませんが、二人の囚人を二つの国家に、「黙秘」を「軍縮」に、「自白」を「軍拡」に置き換えて考えると、軍拡競争は、先の利得行列と質的にはほぼ同一の利害構造を備えていることが分かります。このような応用範囲の広さから、囚人のジレンマは、国際関係論を含む多様な社会科学の研究分野において関心を集めてきました。

では、囚人のジレンマのゲームにおける利害構造のもと、双方が協調を達成しジレンマを回避する方策は存在するのでしょうか。この問題をめぐり実にさまざまな研究がなされてきましたが、ひとつの有力な答えは、この利害構造において二人の行為者が繰り返し相互作用をするならば、協調が合理的な選択になりうるというものです。つまり、ゲームが一度きりではなく、多数回（厳密には無限回）繰り返される状況では、たとえば今日の裏切りは明日の報復へとつながるといった蓋然性が生じることで、双方が協力へと向かう契機が存在することになります。問題は、各行為者が具体的にどのようにゲームをプレイすれば（ゲーム理論の言葉で言うと「どのような戦略をとれば」）、恒常的な協調関係が達成されるのかということです。

この問題に、マルチエージェント・シミュレーションを含むきわめて独創的なアプローチで取り組んだのが、前回も出てきた政治学者ロバート・アクセルロッドです（詳しくは、著書 *The Evolution of Cooperation*, Basic Books, 1984 を参照ください）。彼は、繰り返し囚人のジレンマをプレイするさまざまな戦略を各分野の専門家から募り（最終的には６０戦略以上に及びました）、それらをコンピュータの中で対戦させるという大胆な「実験」を行いました。今回は、このような試みの一端を追体験できるようなモデルを KK-MAS を用いて作っていきましょう。

具体的には、下図のように配置された 50 のエージェントが、上下ペアになって囚人のジレンマゲームを繰り返しプレイするモデルを作っていきます。各ペアは 100 ステップの間ゲームをプレイしたのち、「席替え」をして対戦相手を変えることになります。



エージェントがプレイするゲームの利得行列は、前ページの利得行列を以下のように抽

象化したものになっています。今度は、行列の要素の値が大きいほど高い利得を表していると考えてください。利得の大小関係のみで考えると、二つの利得行列は質的には同一の利害構造を表していることになります。

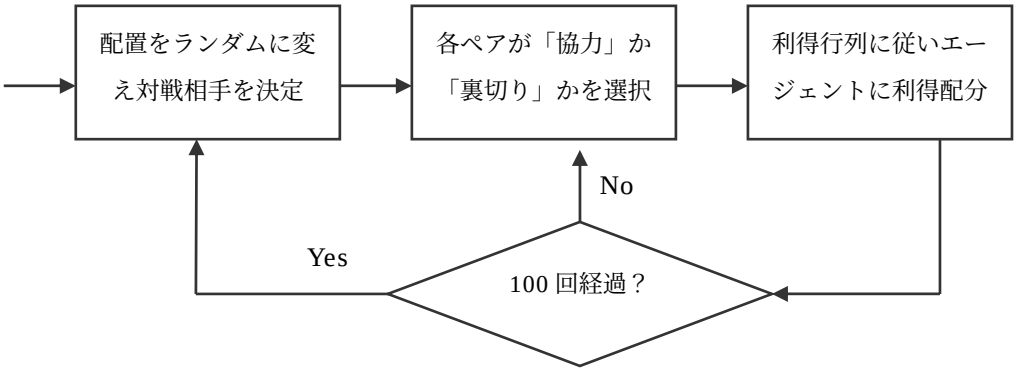
		プレイヤー２	
		協力(C)	裏切り(D)
プレイヤー１	協力(C)	3.0, 3.0	0.0, 5.0
	裏切り(D)	5.0, 0.0	1.0, 1.0

さて、このゲームを各エージェントがどのようにプレイするかですが、今回は、以下の五つの戦略を取り上げ、各戦略をエージェントにランダムに割り当てることにします。エージェントは各々の戦略に従って、各ステップ、協力するか裏切るかを決めていきます。

0. Random 戦略：ランダムに「協力」「裏切り」を選択する。
1. AllC 戦略：相手の手に関わりなく常に「協力」を選択する。
2. AllD 戦略：相手の手に関わりなく常に「裏切り」を選択する。
3. TitForTat（TFT）戦略：最初は「協力」、以後は前回相手が「協力」なら「協力」、「裏切り」なら「裏切り」を選択する。
4. LongMemory 戦略：最初は「裏切り」、以後は過去 10 回の相手の手のうち、「協力」の回数が多いなら「協力」、それ以外は「裏切り」を選択する。

今回のモデルの主眼は、上記の五戦略を相互に対戦させたときに、どの戦略が高い利得を導くのかを検証することです。よく知られているように、アクセルロッドの実験では、TFT 戦略（しっぺ返し戦略）がプレイヤー間の相互協力を導くことで、最も高いスコアを上げました。この結果を再現することはできるのでしょうか。

シミュレーション全体の流れをフローチャートで示しておきます。100 ステップごとに配置転換が入ることで、これまでのモデルと比べ、流れが若干複雑化していることに注意が必要です。



このようなモデルを作りながら、今回は以下の三つの技法および文法事項について学んでいきます。

①記憶と行動：GetHistory 関数再び

今回作るモデルの特徴のひとつは、エージェントの行動パターンが多様で複雑である点です。囚人のジレンマのゲームそのものは、選択肢を二つしか持たない二人の行為者の間の相互作用を形式化したごく単純な意思決定構造になっていますが、これを繰り返しプレイするとなると、「協力」「裏切り」の出し方に関して、無数の行動パターン＝戦略が考えられます。今回はこの行動パターンのうち五つを取り上げるわけですが、中でも TFT 戦略と LongMemory 戦略、とりわけ後者は、過去の相手の行動選択を参照しつつ自分の手を決定するやや複雑な行動パターンになっています。このような行動パターンを KK-MAS においてルール化する上で便利な関数が、すでに前々回、ライフゲームを作った際にも登場した GetHistory 関数です。今回はこの関数と For 文などを組み合わせて用いることで、エージェントの記憶と行動を複雑に結びつけるための手法を学んでいくことになります。

②エージェントの行動のモジュール化：ユーザー定義関数

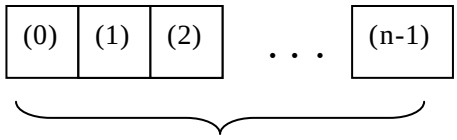
KK-MAS で数々の関数を駆使してきた皆さんにとっては、「関数」が「y=f(x)」形式の数値の間の関係性を表現するものに限定されないことはすでに明らかなはずです。今回ルール化する TFT 戦略や LongMemory 戦略も、「協力」か「裏切り」かという二者択一の選択が、過去の相手の行動選択に依存して決まるという意味で、一種の関数と考えることが可能です。たとえば TFT 戦略は、一期前の相手の行動選択を引数、現在の自分の行動選択を返り値とする関数と見なすことができます。

今回は、実際にこのような形で、TFT戦略とLongMemory戦略を各々ひとまとまりの関数TFT( )およびLongMemory( )として定義する手法を学びます。このようなユーザー独自の関数を、KK-MASが用意している「組み込み関数」（MakeAgtSet関数など）に対して、**「ユーザー定義関数」**と呼びます。ユーザー定義関数は、関数の処理内容を一定の書式に従って、ルールエディタ中のUniv_Step{ }やAgt_Step{ }の外で記述して定義します。これにより、通常の組み込み関数と同じようにルールの中で特定の処理を自由に呼び出すことが可能になります。ユーザー定義関数を用いることで、複雑な処理が並ぶルールの構造をすっきり見やすいものにすることができます。同じ処理を繰り返して用いるようなモデルでは、記述するルールの量と手間を大幅に節約することも可能です。

③配列とデータの入出力

上で述べたように、五つの戦略のパフォーマンスを比較するのが今回のモデルの目的になりますが、各戦略のスコアを出力するために、たとえばScore_RND、Score_AllC…という具合に、逐一変数を作って操作するのは、特に戦略の数が増えると、それだけでかなり

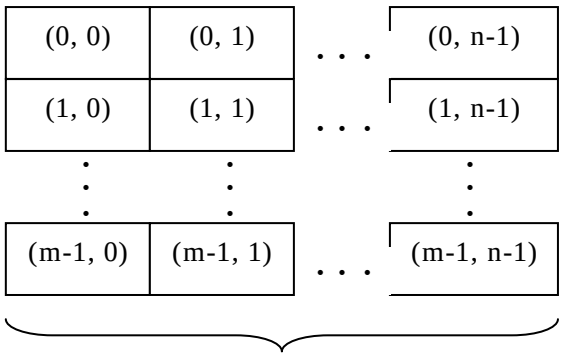
手間のかかる作業になります。このような手間を省く際に大いに活躍するのが、**「配列」**という形式の変数です。これまでの変数は、整数型にせよ文字列型にせよ、一変数につき一つの値しか入れることができませんでした。しかし、配列は、一つの変数でありながら、いくつもの値を「引き出し」に入れることができます（下図参照)。この「引き出し」のことを「配列要素」と言います。たとえば、変数aに 10 の要素があれば、0〜9 までの要素番号を持つ配列要素があります。各要素の値はこの番号を使って、a(2)というような書式で参照します。ここでは、この配列を用いて各戦略のスコアを一括管理することにします。



n 要素の配列変数 a

一次元配列のイメージ

上図のような形式の配列を「一次元配列」と呼びます。これに対して、より複雑な構造を持つ「二次元配列」も存在します。これはちょうど行列のような構造を備えた変数であり（下図参照)、たとえば「A 君の数学の得点」「A 君の英語の得点」…「C さんの数学の得点」「C さんの英語の得点」など二次元形式のデータを一括して管理する際に頻繁に用いられます。ここでは、囚人のジレンマゲームの利得行列を、二次元配列を用いて表現します。二次元配列は、行方向、列方向の二種類の要素番号を持っていて、この値の組み合わせで、どの要素を取り出すのかを指定します。たとえば、b(1, 3)とすると、上から二行目、左から四列目の「引き出し」に入っている要素を参照することができます。



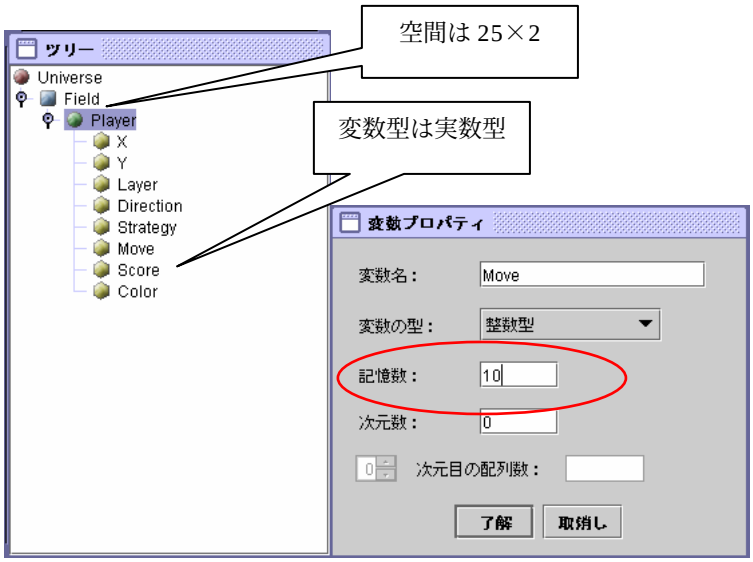
m×n 要素の配列変数 b

二次元配列のイメージ

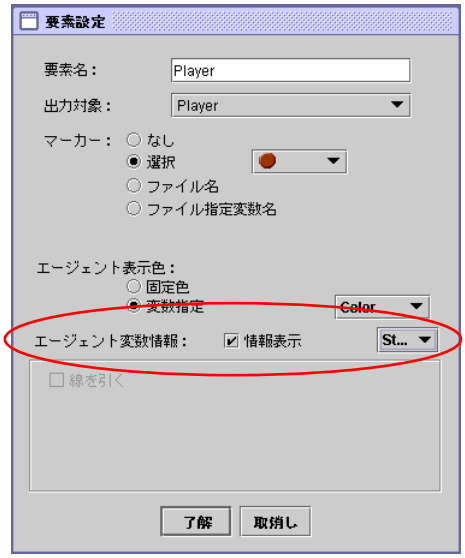
●モデル作り１：エージェントの生成と配置

まず、下図のように、ツリーでモデルの構造をデザインします。空間（以下 Field）は X 方向に 25、Y 方向に 2 の横長の空間にします。ここにエージェント（以下 Player）を追加します。今回はエージェントにいっぱい変数が付いてきます。下図左の Strategy 変数（整数型）は、Player の戦略を 0～4（５ページのものに対応）の整数値で表すための変数、Move 変数（整数型）は各ステップで Player が出す手を代入する変数（0 が「協力」、1 が「裏切り」）、Score 変数（実数型）は Player の総利得を収める変数、Color 変数（整数型）は色表示のための変数です。色は「協力」「裏切り」に連動して変化させることにします。

なお、今回は GetHistory 関数を使って過去の Player の手を最大 10 期前まで参照します。そこで、Move 変数の「記憶数」を、下図右のように「10」に設定しておきます。



マップ出力設定もしておきましょう。要素設定画面は下図のようになります。「エージェント変数情報」で Strategy 変数を選び、戦略の番号がマップに出力されるようにします。



ここでエージェントの生成と配置のためのルール作りです。

[1] 25×2 の二次元空間を Player エージェントで埋め尽くします。Player の戦略は、五つの戦略からランダムに選びます。Player の配置を 100 ステップに一度（1 ステップ, 101 ステップ, 201 ステップ …）ランダムに変更し、対戦相手が変わるようにします。

＞注意するのは、100 ステップごとに Player エージェントをシャッフルすることだけです。この部分のルールは、Univ_Step_Begin に書くことになります。Univ_Init では、下記のように Player を生成するだけです。説明は不要でしょう。

```
Univ_Init{

        Dim i As Integer
        Dim Num As Integer

        // Player を生成（数は空間の幅でコントロール）
        Num = GetWidthSpace( Universe.Field ) * 2
        For i = 0 To Num - 1
                CreateAgt( Universe.Field.Player )
        Next i

}
```

＞Univ_Step_Begin は以下のように書きます。

```
Univ_Step_Begin{

        Dim Set As AgtSet

        // 100 ステップ毎に「席替え」し対戦相手を変える
        If GetCountStep( ) Mod 100 = = 1 Then
                // Player を集めてランダムにばらまく
                MakeAgtSet( Set, Universe.Field.Player )
                RandomPutAgtSetCell( Set, False )
        End If

}
```

こう書くことで 1, 101, 201…ステップに配置ルールが実行される

条件文の「GetCountStep( ) Mod 100 = = 1」がポイントになります。それ以外はエージェントをランダムに配置する見慣れたルールです。GetCountStep 関数（引数なし）はシミュレーションのステップ数を取得するための関数であり、Mod は割り算の余りを求めるための演算子になります。この二つを組み合わせて利用することで、1 ステップ、101 ステップ、201 ステップ…においてエージェントの配置をランダムに変更する処理を実行することができます。

＞Agt_Init のルールはとても簡単です。配色の部分はあってもなくても構いません。

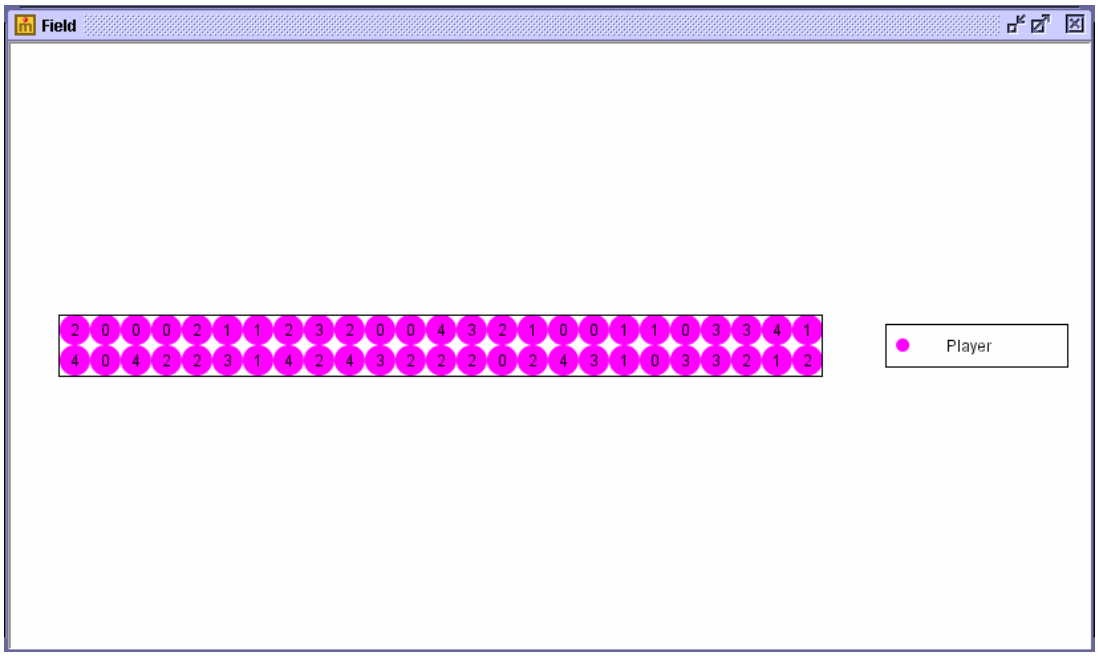
```
Agt_Init{

        // 五つのうちからランダムに取るべき戦略を決定
        My.Strategy = Int( 5 * Rnd( ) )

        // 初期色はとりあえず紫
        My.Color = Color_Mazenta

}
```

実行ボタンを押すと下図のような画面が出てくるでしょうか。しばらく実行するとエージェントの戦略番号の配置ががらりと変化するはずです。



●モデル作り２：エージェントの行動選択のルール

次に、冒頭のフローチャートにおいて二番目のブロックに書かれた処理をルール化していきます。

[２] 各ステップ、Player は、上下ペアになって囚人のジレンマゲームをプレイし、各々の戦略が定める規則に従って相手に協力（Move=0）するか裏切る（Move=1）かを決定します。

＞まず Agt_Step に下記のようなルールを書きましょう。

```
Agt_Step{

        Dim One As Agt
        Dim Opponent As Agt
        Dim Set As AgtSet

        // 同じ列のプレイヤーを対戦相手として取得
        MakeOneAgtSetAroundOwnCell( Set, 1, Universe.Field.Player, False )
        For Each One In Set
                If One.X = = My.X Then
                        Opponent = One
                End If
        Next One

        // ↓戦略に従って協調か裏切りかを決める

        // Random 戦略（Round 関数は四捨五入する関数）
        If My.Strategy = = 0 Then
                My.Move = Round( Rnd( ) )

        // AllC 戦略
        ElseIf My.Strategy = = 1 Then
                My.Move = 0

        // AllD 戦略
        ElseIf My.Strategy = = 2 Then
                My.Move = 1
```

等確率で 1 か 0 を出すための小技

```
        // TFT 戦略（内容は Agt_Step の外で定義）
        ElseIf My.Strategy = = 3 Then
                My.Move = TFT( Opponent )

        // LongMemory 戦略（内容は Agt_Step の外で定義）
        ElseIf My.Strategy = = 4 Then
                My.Move = LongMemory( Opponent )

        End If

        // 手に応じて変色
        If My.Move = = 0 Then
                My.Color = Color_Cyan
        Else
                My.Color = Color_Mazenta
        End If

}
```

TFT 戦略に従った手を返してくれる独自の関数

LongMemory 戦略に従う手を返す独自の関数

若干長いですが、構文的にはシンプルなルールです。

最初に、MakeOneAgtSetAroundOwnCell 関数を使って集めた周囲の Player の中から、X 座標の同じものを探して、対戦相手（Opponent）として選び出しています。これにより、自分より一段上か下の Player との間で囚人のジレンマゲームを行うことになります。

続いて、この相手に対して協力するか裏切るかを、If 文を使って戦略ごとに分岐させた上で決定しています。Random 戦略の行動を記述する際に出てくる Round 関数は実数値を四捨五入するための関数です。0～1 の実数値を返す Rnd( )を引数にすることで、0 か 1 かを等確率で返してくることになります。もちろん、このような小技を使わないで If 文で丁寧に書いてもらっても構いません。

さて、TFT 戦略および LongMemory 戦略の行動決定の部分に出てくる、Opponent を引数とする関数 TFT( )と LongMemory( )とがポイント②で出てきたユーザー定義関数です。上記ルール中では、通常の組み込み関数と全く同じように使用していますが、両関数を機能させるためにはその処理内容や書式を定義しておく必要があります。

まず、TFT 関数を定義しましょう。Agt_Step{ }の外側に以下のようなルールを書き加えていきます。

```
// 一期前の手をそのまま返す TFT 戦略の定義
Function TFT( person As Agt ) As Integer
{

        Dim Action As Integer

        // ラウンド最初は協調
        If GetCountStep( ) Mod 100 = = 1 Then
                Action = 0

        // それ以外は前期の相手の手を返す
        Else
                Action = GetHistory( person.Move, 1 )

        End If

        // 最後に返り値を返すのを忘れずに
        Return( Action )

}
```

冒頭で関数名、引数とその型、返り値の型とを定義する

100 ステップごとに相手が変わるのでこのようにする

一期前の相手の手を参照

一般に、ユーザー定義関数は「Function」で始まる下記の書式で定義します。TFT 関数は、相手の Player をまるまる引数とし、自分が出す手を 0（協力）か 1（裏切り）かの整数値で返す関数です。そこで、冒頭で「Function TFT(person As Agt) As Integer」というように、引数をエージェント型、返り値を整数型として定義してやります。定義する際に引数の変数（上では person、これを「仮引数」と言います）に用いる名称は任意です。

```
Function 関数名( 引数 As 型, ・・・) As 返り値の型
{
        実行したいルール

        Return (戻り値)
}
```

関数の処理内容は{ }の中に挟んで書きます。TFT 戦略は、対戦最初は「協力」、以後は前期の相手の手を今期の自分の手にする戦略になっているので、その内容を上記のようにル

ール化するわけです。相手の前期の手を参照するのに、前々回用いた GetHistory 関数を使用しています。最後に返り値を Return 文で返すのを忘れないようにしてください。

ちなみに、返り値が必要のない関数（これを「サブルーチン」と言います）を作りたい場合もあるかもしれません。この場合には、冒頭の「Function」を「Sub」と置き換え、Return 文を省略すれば、定義することができます。

続いて、LongMemory 関数を定義します。今度はルールの中身に注目しましょう。

```
// 過去十期の相手の手から出す手を決める LongMemory 戦略の定義
Function LongMemory( person As Agt ) As Integer
{
        Dim i As Integer
        Dim Steps As Integer
        Dim Num_C As Integer
        Dim Action As Integer

        // ラウンド最初は裏切り
        If GetCountStep( ) Mod 100 = = 1 Then
                Action = 1

        // それ以外は過去の相手の傾向を分析して手を決める
        Else

                // ↓まずラウンド開始からの経過ステップをチェックして…

                // 10 ステップ以上経過なら 10 ステップ分の記憶を検索
                If GetCountStep( ) Mod 100 - 1 >= 10 Then
                        Steps = 10

                // それ以外は経過ステップ分だけ記憶を検索
                Else
                        Steps = GetCountStep( ) Mod 100 - 1

                End If
```

TFT 関数と同じように冒頭で書式を定義

過去何ステップ遡るかを指定する変数

対戦開始から 10 ステップ以上経過していない場合は、経過ステップだけ記憶を遡るようにする

```
                // 過去の協調回数を相手の手の記憶を遡りながらカウント
                Num_C = 0
                For i = 1 To Steps
                        If GetHistory( person.Move, i ) = = 0 Then
                                Num_C = Num_C + 1
                        End If
                Next i

                // 協調回数の方が多いなら協調
                If Num_C > Steps - Num_C Then
                        Action = 0

                // それ以外は裏切り
                Else
                        Action = 1

                End If

        End If

        // 最後に返り値を返すのを忘れずに
        Return( Action )

}
```

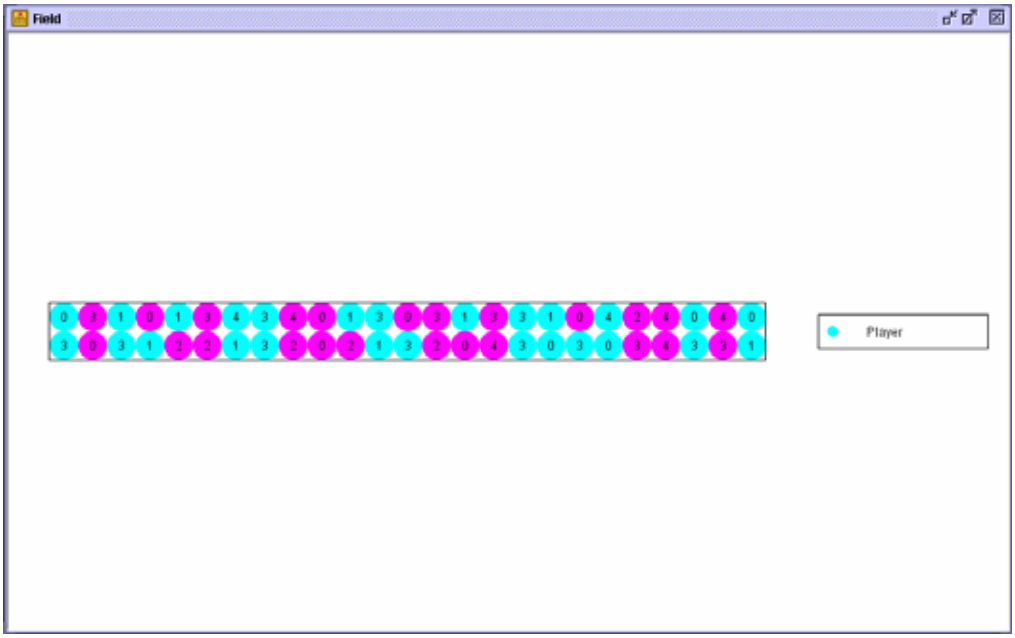
For 文と GetHistory 関数を組み合わせて過去を順々に遡る

LongMemory 戦略は、対戦 1 ステップ目は「裏切り」、以後は過去 10 ステップの範囲で相手の手の出し方をチェックし、「協力」が多いなら「協力」を選ぶ、かなり複雑な戦略です。ポイント①で述べたように、このような戦略をルール化する上で GetHistory 関数の高度な使い方が必要になります。

前々回述べたように GetHistory 関数の第二引数は、何期前の値を返すのかを指定する整数値です。ここで注意する必要があるのは、この引数を実際のシミュレーションの経過ステップを越える値にしてしまうと、エラーが起きてしまうという点です。つまり、過去 10 ステップの記憶を遡ると言っても、シミュレーションがたとえば 5 ステップしか経過していない段階では、実質過去 5 ステップ分の値しか参照できないということです。そこで、このような問題を回避するため、上記のルールでは、過去何ステップ分の値を参照するのかを Steps という変数で指定するものとし、「If GetCountStep( ) Mod 100 - 1 >= 10 Then」で始まる条件文でこの変数の値を決定しています。

このような備えをした上で、For 文と GetHistory 関数を組み合わせて、過去の相手の手の出し方を、一期前から最大で十期前まで順々にチェックしていきます。「GetHistory(person.Move, i)」という具合に、For 文の参照変数 i を GetHistory 関数の第二引数にすることで、こうした処理が可能になります。あとは相手の「協力」の回数をカウントして、その数に応じて「協力」か「裏切り」かを決めて返せばいいだけです。

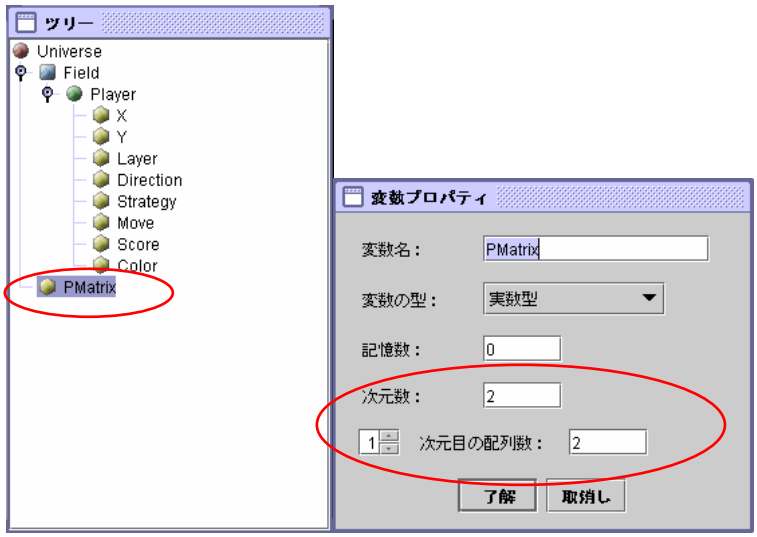
ここまで書き上がった段階でモデルを実行すると、下図のように Player が対戦相手を変えながら、各々の戦略に従って囚人のジレンマゲームをプレイし始めるはずです。



●モデル作り３：利得の配分

各ステップにおける「協力」と「裏切り」の手の組み合わせに応じて、Player に利得を配分すれば、モデルの中身は完成します。手の組み合わせと利得との対応関係は、5 ページの利得行列に示したとおりです。この利得行列は、ポイント③で出てきた二次元の配列を利用することで、ひとつの変数として表現することが可能です。というのも、利得行列をよく見れば分かるように、自分と相手の手の組み合わせに対する利得は、二人のプレイヤーの間で全く同一であるからです。

そこで、次ページ図左のように Universe 直下に利得行列を表わす変数 PMatrix を追加します。配列変数をツリーで定義する場合、図右のように、変数プロパティの「次元数」で配列の次元数を、「配列数」で各次元の要素数を指定することになります。ここでは２×２の二次元配列を作るので、「次元数」は「２」、「１次元目の配列数」「２次元目の配列数」はともに「２」にします。変数型は実数型にしておいてください。

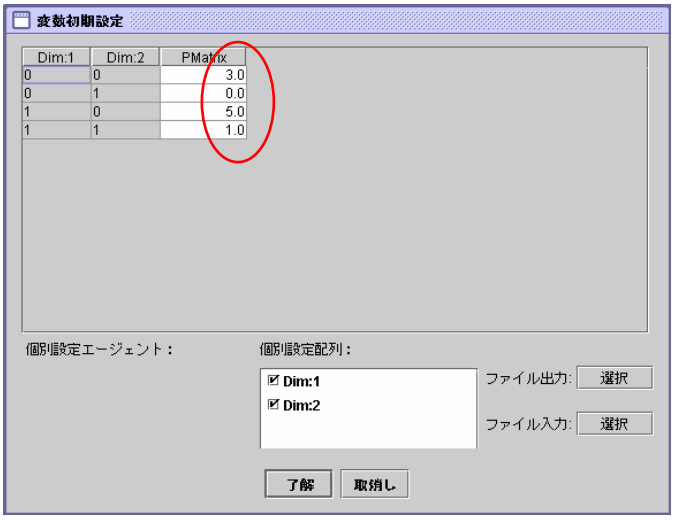


このように定義した二次元配列 PMatrix に下図に示すような意味を与えます。つまり、一次元目（行）の要素番号 0, 1 を自分の「協力」「裏切り」と、二次元目（列）の要素番号 0, 1 を相手の「協力」「裏切り」とそれぞれ対応付け、たとえば PMatrix(0, 1)は、自分が「協力」、相手が「裏切り」を選択したときの利得 0.0 を表わしていると考えるわけです。

		相手	
		協力(0)	裏切り(1)
自分	協力(0)	3.0	0.0
	裏切り(1)	5.0	1.0

PMatrix(0, 1) で参照

実際に、変数 PMatrix にこうした意味を持たせるため、配列の各要素に数値を入力しておきます。PMatrix を選択した状態で右クリックするか、メニューバーで「設定」をクリックして、「初期値設定」を選びます。「変数初期設定」ダイアログで下図のように数値を入力してください。「Dim:1」が行の要素番号、「Dim:2」が列の要素番号を表わしています。



ここまで準備をした上で、Player に利得を配分するルールを書きます。

[3] 空間左端から順次 Player のペアをチェックしていき、両者の「協力」「裏切り」の組み合わせから各々が受け取る利得を算出し、Player の Score 変数に加えていきます。

＞全ての Player が一通り行動選択を行った後なので、Univ_Step_End にルールを書き込んでいきます。

```
Univ_Step_End{

        Dim i As Integer
        Dim P1 As Agt
        Dim P2 As Agt
        Dim Set As AgtSet

        // For 文を用いて左端のペアから順次スコアを配分
        For i = 0 To GetWidthSpace( Universe.Field ) - 1

                // ↓上下段の Player を取得
                // 各座標点にいる 1 Player のみから成る集合を作り、GetAgt 関数で取り出す
                MakeOneAgtSetAroundPosition( Set, Universe.Field, i, 0, 0, 0.1,
Universe.Field.Player )
                P1 = GetAgt( Set, 0 )
                MakeOneAgtSetAroundPosition( Set, Universe.Field, i, 1, 0, 0.1,
Universe.Field.Player )
                P2 = GetAgt( Set, 0 )

                // 利得行列を使ってスコアを配分
                P1.Score = P1.Score + Universe.PMatrix( P1.Move, P2.Move )
                P2.Score = P2.Score + Universe.PMatrix( P2.Move, P1.Move )

        Next i

}
```

指定した座標点の周囲のエージェントを集める関数

集合に 1 エージェントしかいないので番号 0 で取り出せる

Player の手の組み合わせで配列 PMatrix を参照し利得を取り出して加算

空間左端からペアを順次チェックしていくのでルール全体が For 文によって構成されています。

For 文の中でまず行っているのは、X 座標 i（0, 1, 2, ...）を共有する上下の Player のペア（P1 と P2）を取り出すことです。KK-MAS では、エージェントはまずエージェント集合に納めてから取り出すのが原則になっているので、やや迂遠な方法を用いています。新たな関数「MakeOneAgtSetAroundPosition」が出てきますが、この関数は、

MakeOneAgtSetAroundPosition(エージェント集合型変数, X 座標, Y 座標, レイヤー, 視野, エージェント種別)

という書式で、指定した座標点の周囲にいるエージェントを集合に納めて返してきます。三番目の引数「レイヤー」についてはここでは説明を省きますが、普通は「0」を指定します。これまで用いてきた関数「MakeOneAgtSetAroundOwn」のように特定のエージェントではなく、特定の座標点を取り囲むエージェントを集めたいときに使う関数です。

上記ルールでは、この関数の第五引数「視野」が 0.1 になっています。これにより、指定した座標点（(i, 0)および(i, 1)）の周囲 0.1 の範囲にいるエージェント、つまり事実上当該座標点に配置されている１エージェントだけがエージェント集合型変数 Set に格納されることになります。前回説明したように（資料 12 ページ参照)、エージェント集合におけるエージェントには 0 から始まる通し番号が割り振られているので、格納された Player の番号は 0 になります。「GetAgt(Set, 0)」とすれば、この Player を取り出せるわけです。

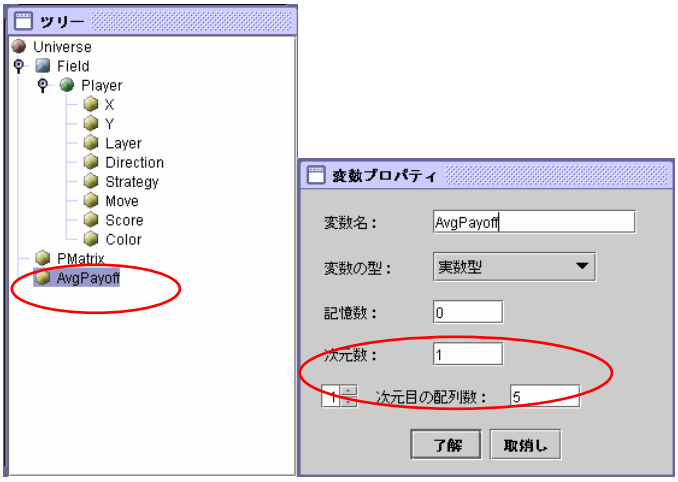
あとは、こうして取り出したペア P1 と P2 の手の組み合わせ（P1.Move と P2.Move）から利得を求めて、各々の Score 変数に加えるだけです。先に説明した配列変数 PMatrix の意味から、「PMatrix(P1.Move, P2.Move)」がこの手の組み合わせに対する P1 の利得、「PMatrix(P2.Move, P1.Move)」が P2 の利得を表わしていることは理解できるはずです。

●モデル作り４：五戦略の平均スコアの出力

最後に、五つの戦略を取る Player たちが１ステップ、１エージェントあたり平均どれだけの利得を上げているのかを戦略ごとに出力させて、モデルを完成させましょう。出力形式は「棒グラフ」にします。手順は「時系列グラフ」の場合と全く同じです。

最初に Universe 直下に出力のための変数を作りますが、ポイント③で述べたように、今回は出力値が五つに及ぶので、一次元配列を使って一括して管理するのが便利です。そこで、次ページ図左のように Universe に実数型の出力変数 AvgPayoff を追加し、プロパティで図右のように設定します。PMatrix と違い、今度は一次元配列なので、「次元数」は「１」とし、五つの戦略の平均スコアを代入するので、「1 次元目の配列数」は「５」とします。

配列変数 AvgPayoff の要素番号 0～4 は、五つの戦略の番号に対応していると考えます。たとえば、AvgPayoff(3)には、TFT 戦略の平均スコアが代入されることになります。



[４] 各ステップ、全 Player の Score をチェックし、１ステップ１プレイヤーあたりの平均スコアを、五つの戦略ごとに分けて算出し、AvgPayoff(0)～AvgPayoff(4)に代入していきます。

＞ルールは、利得配分の際と同様、Univ_Step_End に記述します。まず、以下のように四つの変数（赤字の部分）を新たに宣言します。

```
Univ_Step_End{

        Dim i As Integer
        Dim P1 As Agt
        Dim P2 As Agt
        Dim Set As AgtSet
        Dim Str As Integer
        Dim One As Agt
        Dim Sum( 5 ) As Double
        Dim Num( 5 ) As Double
```

実数型の一次元配列変数を宣言

一次元配列変数をルールエディタの中で一時的な変数として宣言する場合、上記のように「変数名(要素数)」という書式を用います。二次元配列変数の宣言は、「変数名(１次元目の要素数, 二次元目の要素数)」になります。

平均値を求める部分は、下記の通りです。利得を配分するルールの下に書き加えていきます。

```
// ↓各戦略の平均スコアを計算

// 配列の初期化
For i = 0 To 4
        Sum( i ) = 0
        Num( i ) = 0
Next i

// 各 Player をめぐって戦略毎の総利得と人数をカウント
MakeAgtSet( Set, Universe.Field.Player )
For Each One In Set

        // 戦略を参照
        Str = One.Strategy

        // 当該戦略のスコアと人数を加算
        Sum( Str ) = Sum( Str ) + One.Score
        Num( Str ) = Num( Str ) + 1

Next One

// 最後に頭数×ステップ数で割ってスコアの平均値を算出
For i = 0 To 4
        Universe.AvgPayoff( i ) = ( Sum( i ) / Num( i ) ) / GetCountStep( )
Next i
```

配列の値をまとめて参照・操作する際には For 文を使う

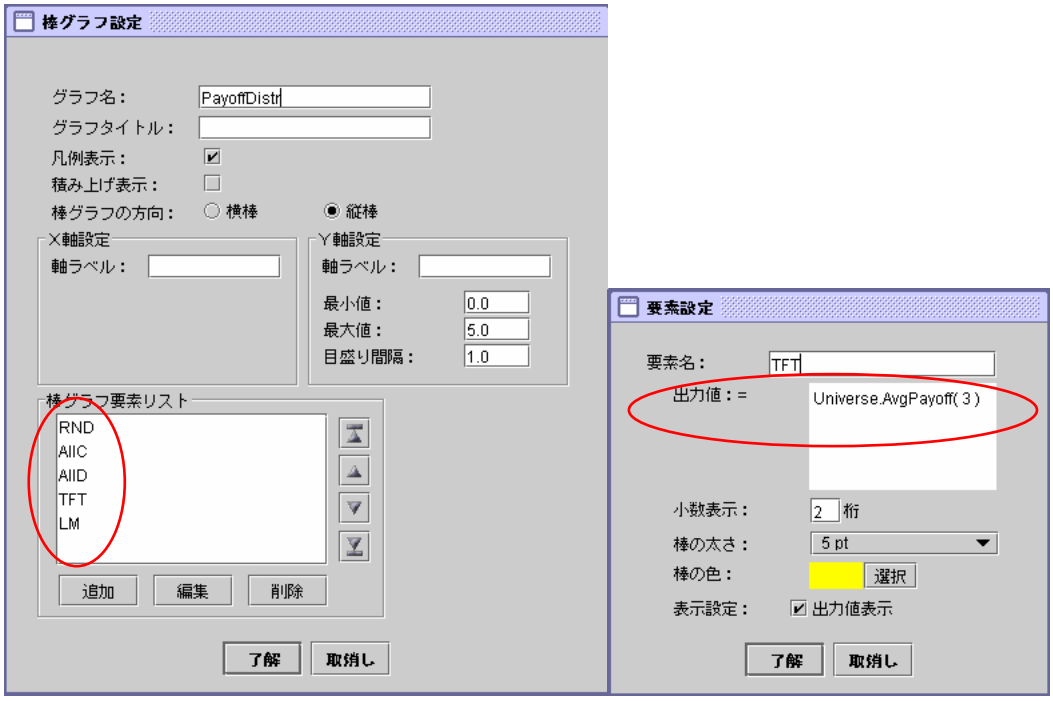
戦略の番号で対応する総利得と人数を参照

上記ルール全体の流れは、For Each 文を使って、それぞれの戦略を取る Player の Score を加算したものを配列変数 Sum に、戦略ごとの Player の人数を配列変数 Num に代入した上で、最後に前者を後者で除したものを、さらにステップ数で割って、各戦略の平均スコアを算出するというものです。Sum と Num の要素番号は、AvgPayoff 変数と同様、五つの戦略の番号に対応するものになっています。

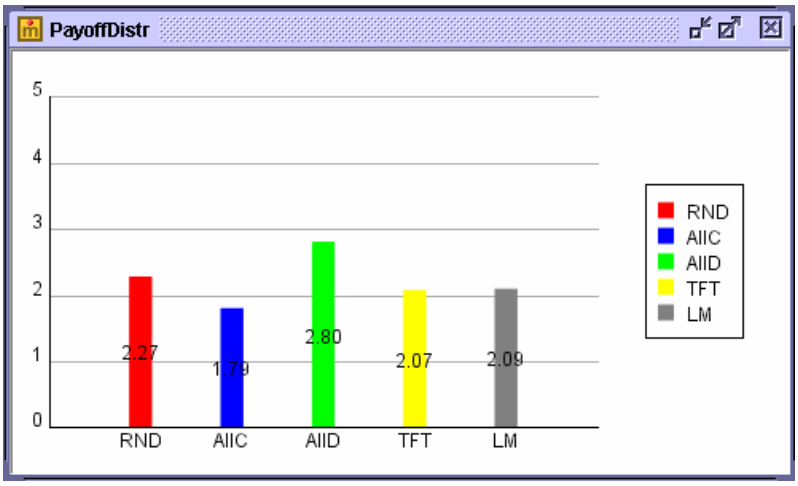
配列を初期化する部分、および最後に平均値を求める部分で、For 文を使って五つの配列要素全てを一括して参照・操作する処理が出てきます。配列は、このように For 文と一緒に用いることで、別々に変数を設けて操作する場合よりもはるかに効率的でスマートな処理を可能にしてくれます。

ここまで来れば、あとは五つの戦略の平均スコアを棒グラフで出力するように設定するだけです。下図のように、出力設定の「出力項目リスト」において、「追加する出力種類」プルダウンから「棒グラフ」を選んで「追加」をクリックします。

続いて出てくる「棒グラフ設定」ダイアログで、グラフ名等を入力した上で、「棒グラフ要素リスト」に順次五つの出力要素を追加していきます（下図左）。要素設定は、TFT 戦略の平均スコアを例に取ると下図右のようになります。「出力値」の部分に「Universe.AvgPayoff(3)」という具合に各配列要素を記述していくことになります。「出力値表示」のチェックボックスをオンにすると、棒グラフと一緒に変数値も出力することができます。



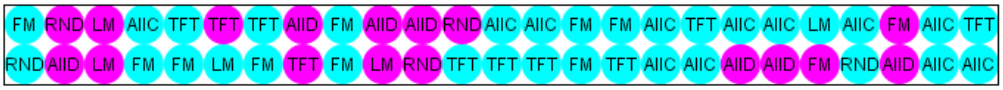
実行ボタンを押して五本の棒グラフが並んで出力されれば、本日の課題は見事クリアです。九回の講義も終わりです。本当にお疲れ様でした！



●宿題

[5] アクセルロッドの囚人のジレンマのトーナメントに参加した戦略のうち、経済学者 J.W. フリードマンが提案した戦略も比較的よく知られています。それは、初回で「協力」を出し、相手が一度でも裏切ったら以後ずっと「裏切り」を出し続けるという戦略です。この戦略を関数化し、今回作ったモデルに組み込んでみましょう。つまり、第六の戦略を登場させるということです。もちろん、ほかにも戦略をいっぱい追加してもらっても構いません。

[6] Player に文字列型変数 Label を追加して出力し、下図のように各 Player の戦略を番号ではなく、戦略名の略称の文字列で表示できるようにしてみましょう。文字列型の一次元配列を Universe に作って利用すれば、出力のためのルールは一行で書けてしまいます。



[7] 最後に、前回学んだ連続実行の機能等を活用して、繰り返しシミュレーションを行ってみましょう。多くの試行で高いパフォーマンスを示す戦略はいったいどれでしょうか。