『はじめての78Kマイコン』付属78K0Sマイコン・ボードを利用した学習型ソーラーカー

奈良県
佐藤 和宏 様

はじめに

今回紹介するのは、フォトセンサと太陽電池を搭載し、フォトセンサからの入力で太陽電池の発電による電力(出力電圧)が大きくなるような動きを自分自身で試行錯誤しながら学習していく、学習型ソーラーカーです。構造は単純ですが、比較的新しい制御理論である知的制御のひとつ、強化学習の仕組みを取り入れたという、なかなか本格的なインテリジェント・カーと言えるでしょう。

写真1は今回製作したソーラーカーの概観、**図1**はソーラーカーの回路図です。Solarと書いてあるのが太陽電池、モーターと接続されているのはモーターのON/OFF制御用のパワーMOSFETです。

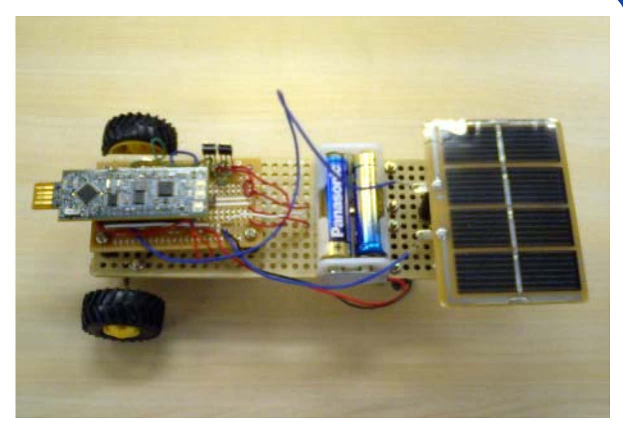

写真1

太陽電池で走行できれば良かったのですが、今回使用している太陽電池ではモーターを回すだけの電力が取れなかったので、マイコンで出力電圧を検出するだけにしています。

光を当てると、フォトセンサの出力を検出しながら太陽電池の出力電圧がなるべく大きくなるような移動方法を自分で見つけていきます。どのように動くかはプログラムを組んだ本人にもわかりません。ソーラーカー自身がどのように学習をしてどのような挙動を示すようになるのか、実際に工作をして楽しんでみてください。

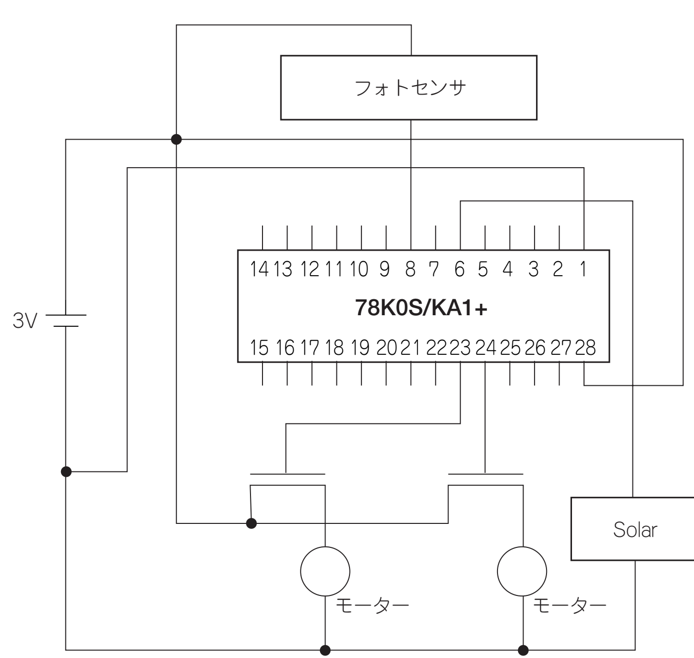

図1　ソーラーカーの回路図

制御についておさらい

今回のソーラーカーの動作の目標は太陽電池が最大電力を発生するようにモーターを駆動するということにあります。これは入力情報による車の制御ということに他なりません。それではまず機械の制御について簡単におさらいしておきましょう。

シンプルな制御

ごくシンプルな制御系の例として**図2**を挙げます。コンベアを流れてきた原料を上下するカッターで切断するというものです。図でもわかるとおり、このコントローラは単純に時間や信号出力などの管理を行うだけで制御対象となるコンベアやカッターの動作状態の確認は行っていません。

コンベアがステッピングモーターのように、パルス数で移動距離が決まるようなものであれば、一定のパルス分だけ移動させてカッターが上下する動作を繰り返させるだけでも実用上問題なく利用できるでしょう。

次にもう少し手間のかかる例として**図3**のように、ヒーターの制御で温度をコントロールする水温制御装置を考えてみます。もし、この装置が完全に理想状態にあり、不確定な要素がまったく無ければ、ヒーターの出力をどのくらいにすれば、水温が何度で安定するかということが決まってくるでしょう。ヒーター出力をどのように変化させれば水温がどのように変化するのかも簡単に決まりますので、先ほどのカッターと同様に「○秒間ヒーターをONにして、○秒間OFFにする」ということを繰り返すだけでも良いかもしれません。しかし、給湯装置や熱帯魚用の水槽の温度制御などを想像すればわかるとおり、現実には水の量やヒーターの仕様、容器の特性、周囲温度、測定の誤差や経時変化など、不確定な要素が山ほどあり理想状態には程遠いものになっています。理想状態を元にヒーター出力を決めても予定どおりの温度で安定するということは期待できません。

そこで、水温を測定して、設定値に近づくように制御するという方法が考えられます。この方式のコントローラにすれば**図3**のように設定値(SV : Setting Value)と制御対象の測定値(PV : Process Value)、制御出力値

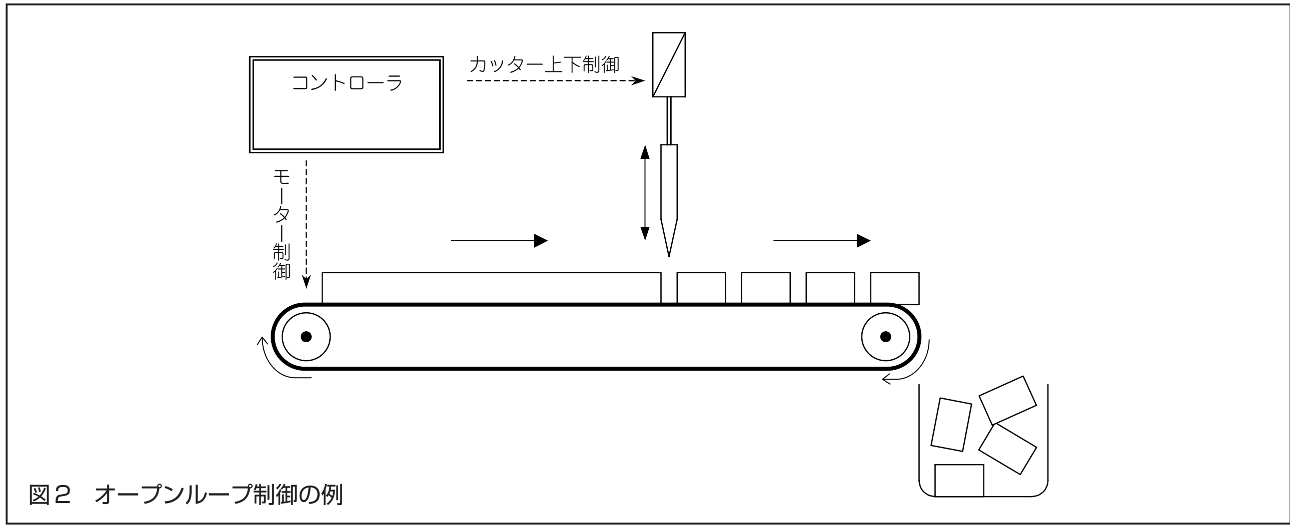

図2　オープンループ制御の例

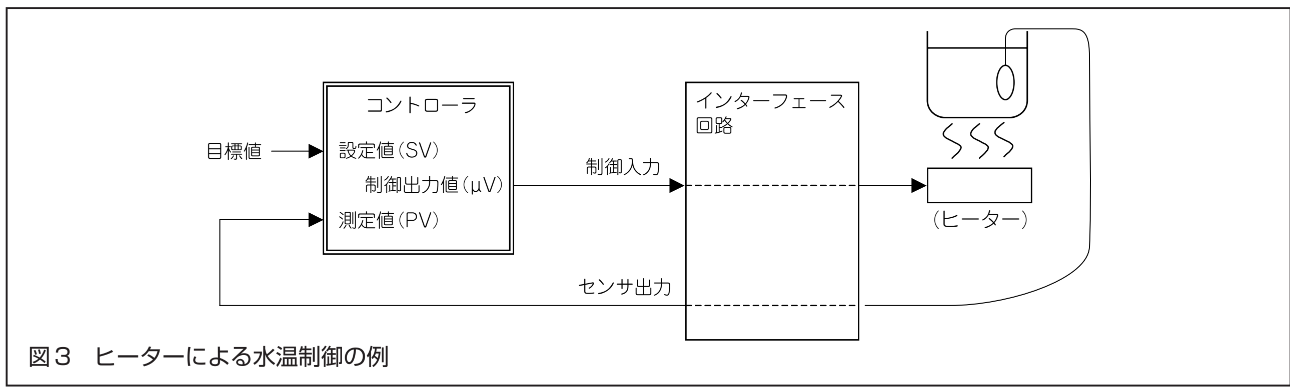

図3　ヒーターによる水温制御の例

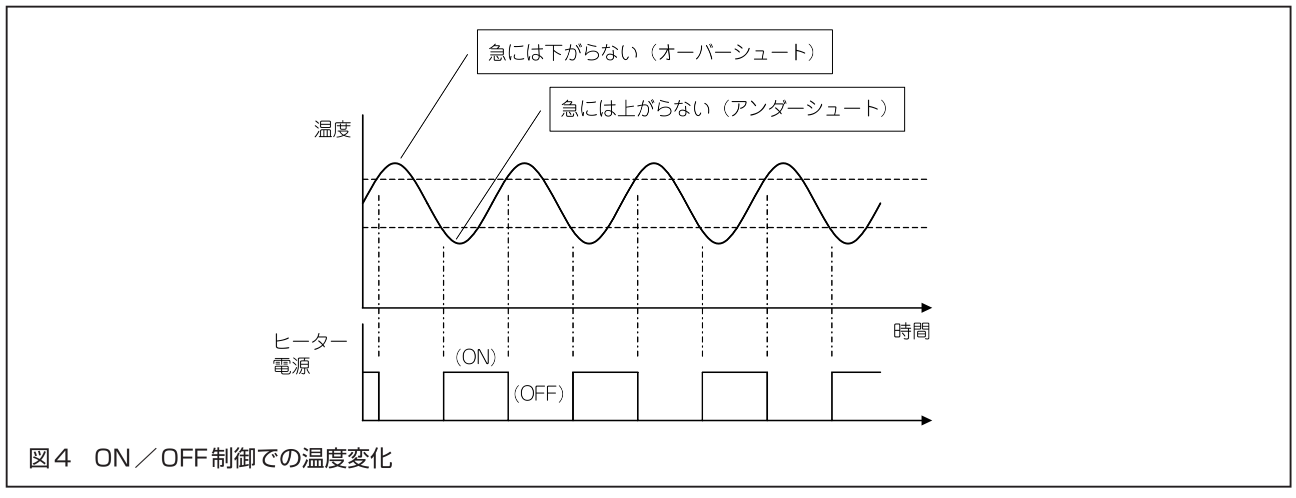

図4　ON／OFF制御での温度変化

(MV：Manipulating Value)の3要素を使って水温制御を行えます。

コントローラはSV値とPV値の差、すなわち設定値と実際の制御対象の測定値の差分を使い、MV値を生成するわけです。この例でいけば、設定温度よりも水温が低ければ水温が上がるようにヒーター出力を大きくし、水温が高ければヒーター出力を小さくします。

図を見ると信号の流れに温度センサによるフィードバックがかかり、リング状(ループ)になっていることがわかります。これをクローズド・ループと呼ぶのに対し、先ほどの切断機のようにフィードバックを持たず、信号を出力しっぱなしのものをオープン・ループと呼ぶこともあります。

単純ON/OFF制御

このような制御ループを持った装置の場合、PV値とSV値の情報からどのようにMV値を生成するかということがポイントとなってきます。もっとも単純なのは次のようなものでしょう。

- PV値がSV値よりも小さい(つまり、設定温度よりも水温が低い)ならヒーターをON
- PV値がSV値より大きいなら(水温が高くなれば)ヒーターをOFF

熱帯魚用の水槽や電気炬燵のサーモスタットのような動きです。現実にはSV値ぴったりで判定するとON/OFFを頻繁に繰り返してしまう可能性がありますので、ある程度の幅を持たせて一度OFFになったらある程度下がらないとONにならないようにすることが普通です。

温度制御では設定した温度まで早く到達し、外乱などによる変化にもすばやく追従するなど、温度の変動がなるべく少なく、しかも安定しているのが理想です。単純ON/OFF制御は確かに簡単ではありますが、安定度という面ではあまり良いものではありません。

対流などもありますので通常、ヒーターの電源を切ってもすぐに温度は低下しません。温度が次第に緩やかになり、その後低下しはじめるという動きになるでしょう。水温が低下するときも同様で、すぐには上昇に転じず、しばらく下ってから上がってくることでしょう。ちょうど**図4**のように上昇と下降を繰り返すことになるわけです。

PID制御について

単純なON/OFF制御の問題はMV値(制御出力)がONとOFFの2状態しかないことと、SV値(設定温度)とPV値(水温)の取り扱いが、単にSVを超えているか否か程度で判定しているという点にあります。もし、これらをアナログ的な入出力に出来ればもっときめ細かな制御が可能となり、温度変化も小さく抑えられるはずです。

● P制御

まず考えられる改善策は、PV値(水温)がSV値(設定温度)よりも大幅に下回っていればヒーターの出力を大きくし、近づいてきたら小さくするという方法です。先ほどは水温が設定温度より高い／低い程度であったものを、温度が大幅に低いときにはヒーター出力をより大きくし、差が小さくなってきたら出力をそれに応じて引き下げるという

考え方です。

式で表せば

$$MV = \alpha \cdot (SV - PV) + K \quad (\alpha、Kは定数)$$

ということになります。KはSV=PVで安定したときのMV値を決めるものです。このような制御をPropotional(比例)制御、または省略してP制御と呼びます。また、定数αを比例ゲインと呼んでいます。

P制御ではKの存在によるオフセットの発生が厄介な点です。式からもわかるとおり、KはSV=PVになったときのMV値です。水温制御装置でいえば、水温が安定しているときのヒーター出力になるわけですが、当然のことながらこの値は設定する水温や水の量、容器の形状や材質などさまざまな要因で変わってきますので、適切な値を決めるのはかなり面倒なことです。

仮にK値が適切な値よりも大きかったとしましょう。このときSV=PVになってもKの影響で温度が上昇し続けてしまいます。そのうち$\alpha \cdot (SV - PV)$が利いてきて温度を下げる方向に向かい、結果的にこの両者が均衡したところで落ち着きます。このズレの分をオフセットと呼びます。P制御ではこのようなオフセットが発生してしまうのが欠点です。

● PI制御

それでは、オフセットを除去するにはどうしたら良いのでしょうか。ここで人間であればどうするか考えてみましょう。オフセットがあるということは、目的の温度からずれたところで安定しているということです。人間ならば、しばらく様子を見ていつまでも目的の温度にならないようであれば、ずれている分を補正するように調整することでしょう。

これと同じようなことをコントローラで実現するにはどうしたら良いでしょうか。人間の行動を少し機械的に見直してみると、「誤差のある状態が継続していれば」「出力が補正される」ような動きということ、すなわち誤差の累積結果がK値を打ち消すような方向に向かえば良いということです。この、「誤差の累積」を数学的に考えれば、誤差の積分値と言い換えることができるでしょう。

式で示せば$\beta \cdot \int (SV - PV)dt$　(βは定数)ということになります。この式のβを積分ゲインと呼んでいます。P制御にこの積分(Integral)分を追加して

$$MV = \alpha \cdot (SV - PV) + \beta \cdot \int (SV - PV)dt + K$$

としたものをPI制御と呼びます。PI制御のときのKは仮の初期値です。多少不適切な値であっても(たとえばゼロであっても)、積分項によって補正されますので大きな問題にはなりません。つまり、

1) (SV－PV)の絶対値が大きいとき、すなわちPV値がSV値よりも大きく外れているときには比例項が大きく利いてきて、PV値をSV値に接近させる。(P制御と同様)
2) PV値がSV値に近づいて(SV－PV)が小さくなったときには積分項が大きくきいてきて、小さい誤差を更に減らして設定値になるように追い込んでいく。

という二段構えの方法で安定化を図っているわけです。

PI制御では積分記号などが出てきて難しそうに思えるかもしれませんが、実際のプログラムはそれほど難しいことにはなりません。マイコンシステムでは数学的な積分とは異なり、無限に短い時間でデータを取り込むようなことはできません。あくまでも一定時間ごとにPV値をA/Dコンバータで読むだけです。したがって積分とは言っても、実態は単なる加算です。式で表せば

$$\beta \cdot \Sigma ((SV - PV) \cdot \Delta t) = \beta \cdot \Delta t \cdot \Sigma (SV - PV)$$

となります。Δtはサンプリング期間で、通常一定期間ごとにサンプリングしますから、$\beta \cdot \Delta t$は固定値です。つまり、単に毎回計算した(SV－PV)に定数を掛けるだけで良いのです。

● PID制御

P制御のKを補正しようというのがPI制御ですが、αの項も適切な値を決めるのが難しい要素です。たとえば、今考えている水温制御装置であれば、水の量が変われば適切なαの値は変わってきます。周囲の気温が変化した場合や容器の材質や形状が変化したときも適切なαの値は変わってきます。また、制御を乱すような外的作用(外乱)によってPV値に急激な変化があったときにもPI制御では追従が遅れがちです。

やはり人間の行動で考えてみましょう。温度変化があまりにも急激であれば抑制するように、上がり方が遅ければ

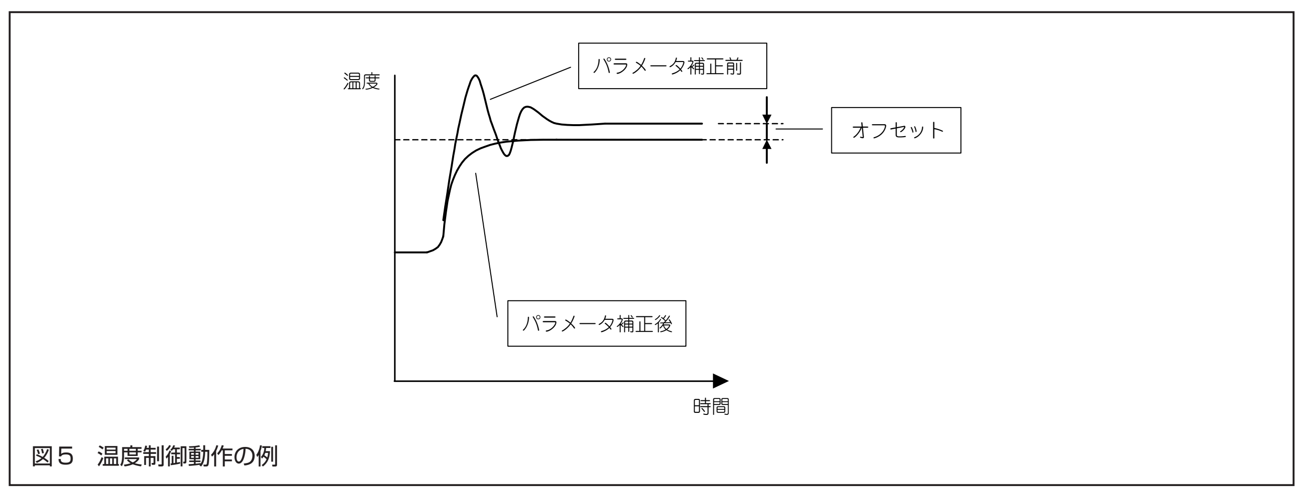

図5　温度制御動作の例

少し増加するように調整することでしょうし、安定しているときに急な変化があればあわてて再調整するでしょう。

数学的に変化が急かどうかを示す指標といえば微分(Derivative)です。これを式で表せば

$\gamma \cdot d(SV-PV)/dt$（γは定数）

という具合になります。ここでγを微分ゲインと呼んでいます。先ほどのPIにこの項を加えると

$MV=\alpha \cdot (SV-PV)+\beta \cdot [(SV-PV)dt+\gamma \cdot d(SV-PV)]/dt+K$

となります。これをPID制御と呼んでいます。積分のときと同様に微分とは言っても数学的に厳密な意味での微分は行いようがないので、前回の(SV－PV)値と、今回の(SV－PV)値の差分、すなわちx回目のSV値、PV値をそれぞれSV(x)、PV(x)と表記すれば、

$\gamma \cdot ((SV(n)-PV(n))-(SV(n-1)-PV(n-1)))/\Delta t$

という具合になります。やはり$\gamma/\Delta t$は定数ですので、微分演算の本体は引き算そのものです。PID制御の場合、急激な変化があればDによって補正しようと働きますし、変化が穏やかであれば、PI制御として働くことになります。

PIDの比例ゲイン、積分ゲイン、微分ゲインを適切に設定してやれば、**図5**のように、スムーズな温度制御が行えます。

知的制御

現在では制御理論もさまざまな発展を遂げて最適制御論、H^{∞}制御理論などが次々に登場しています。これらの中のひとつに「知的制御(Intelligent Control)」と呼ばれるものがあります。「このような場合には出力を大き目にする」という人間の勘や経験に基づくあいまいさを制御アルゴリズムとして取り入れたファジイ制御などをはじめ、ニューラルネットワーク、遺伝的アルゴリズムなどの知的制御は実際にエレベータの速度制御や電車(仙台市の地下鉄南北線など)などにも利用され、乗り心地の良い運転が行われています。

今回採用した強化学習もまた知的制御のひとつです。知的制御はそれぞれ非常に興味深いものですので、興味のある方は専門書にあたってみてください。ここでは今回のソーラーカーに採用した強化学習と実際のプログラムの動作に絞って見ていくことにします。

強化学習（Reinforcement Learning）

強化学習の基本的な構成は**図6**のようになっています。図でもわかるとおりコントローラと制御対象の間で信号のループができていることはPID制御などと同じですが、行動／状態／報酬という、名称になっています。また、強化学習ではコントローラ側をエージェント、制御対象を環境と呼んでいます。

エージェントの動作の考え方も変わっています。PID制御は制御対象の測定値と設定値の誤差が最少になるように制御するという方法でしたが、強化学習では現在の環境の状態(s ： Status)と報酬(r ： Reward)が与えられ、エージェントは、状態sを見ながら、報酬rが最大になるように、どのような行動aをとるかを決めていくのです。制御というよりも、仮想の知能ロボットのようなものをイメージしたほうが良いかもしれません。

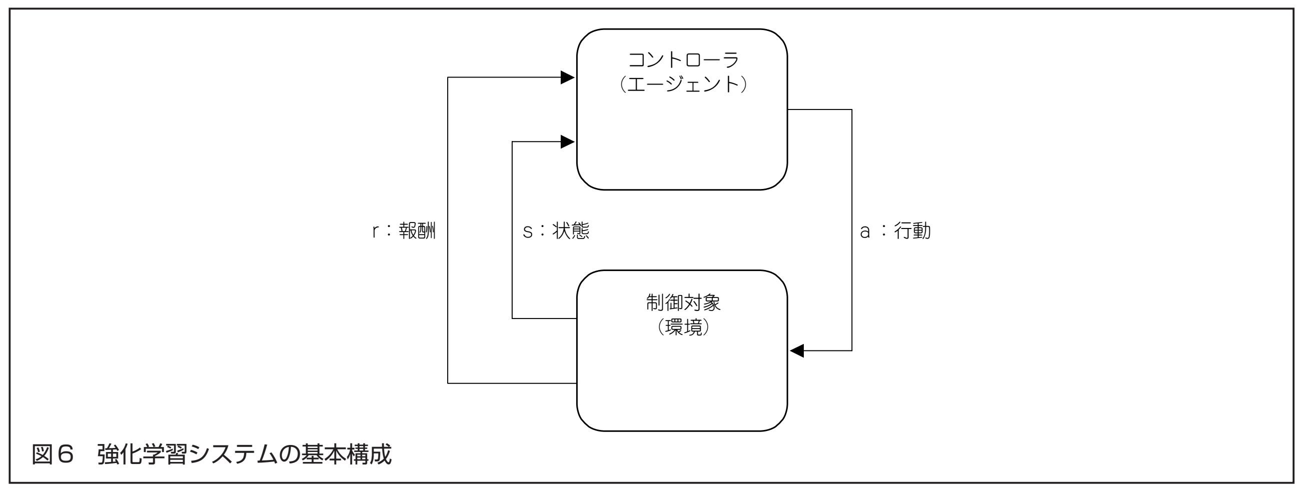

図６　強化学習システムの基本構成

たとえば、棒を台の上に立てて、倒れないように台を動かす場合を考えてみます。このとき、棒の現在の傾きの方向や速度、現在の台の移動方向、速度などといった状態を示すパラメータが状態ｓ、安定して立っている状態に近いか否かを示すのが報酬ｒとすれば、ｓを利用して、ｒが最大になる、すなわち安定して立つ状態になるように行動ａを選択していくという具合です。

ここでポイントになってくるのが行動ａの決定方法です。もし制御対象が比較的単純なもので、人間がどのような行動ａを取れば良いのかがすぐわかるなら話は簡単です。人間が常に正しい行動を教えてエージェント側はそれを記録していけば良いわけです。あるいは、最初から正しい行動をとるようにプログラム中に記述してしまえば良いのです。

このようにあらかじめ正解がわかっている人に正しい行動を決めてもらいながら学習していく方法を教師付き学習（Supervised learning）と呼んでいます。

ところが、制御対象によっては人間でも最適解がよくわからないということも多々あります。たとえば、手足のついたロボットを前に歩かせる場合でも、どのように制御するのがもっとも効率の良い方法であるかということは簡単には答えが出せません。また、アップ＆ダウンの多いコースを作り、模型自動車をＡ地点からＢ地点に走らせるには、どのようなルートを通るのが一番早いかという程度のことでも簡単には割り出せないでしょう。

このような場合には教師の存在には頼らず、エージェント自らが試行錯誤しながらその結果を見て最適な解を見つけていくという方法が考えられます。これが強化学習の基本的な考えです。より良い状態にあるのか否かを示す指標として報酬ｒが導入されています。エージェントが自ら色々なパターンを試行錯誤していくことで、人間でも思いつかなかったような良い解答を得られる可能性があります。

強化学習型ソーラーカー

今回の学習型ソーラーカーでは

- 状態ｓ入力はフォトセンサの値で、明るさの前回値からの変化量により０～５の６段階
- 報酬ｒ入力は太陽電池の発電量（電圧）の変化値、変動値により３値（－50、0、+50）をとる
- 行動ａは右回転、左回転、直進の３つの中から選択

という設定にしています。これらを使って、フォトセンサの明るさの変化から、太陽電池の発電量がより大きくなるような行動パターンを学習していきます。人間の場合でも、過去に色々なことを試してきた経験から、これは結果的に上手くいった／最悪の結果になった、少し改善／改悪されたなどの経験を積んでいきますが、これと似たようなことをプログラムで実現します。

強化学習では次に取るべき行動を選択するための情報として過去に行ってきた行動の結果を利用するのがポイントになります。この手法にも色々なものが考えられていますが、今回はsarsaと呼ばれるアルゴリズムを採用しています。

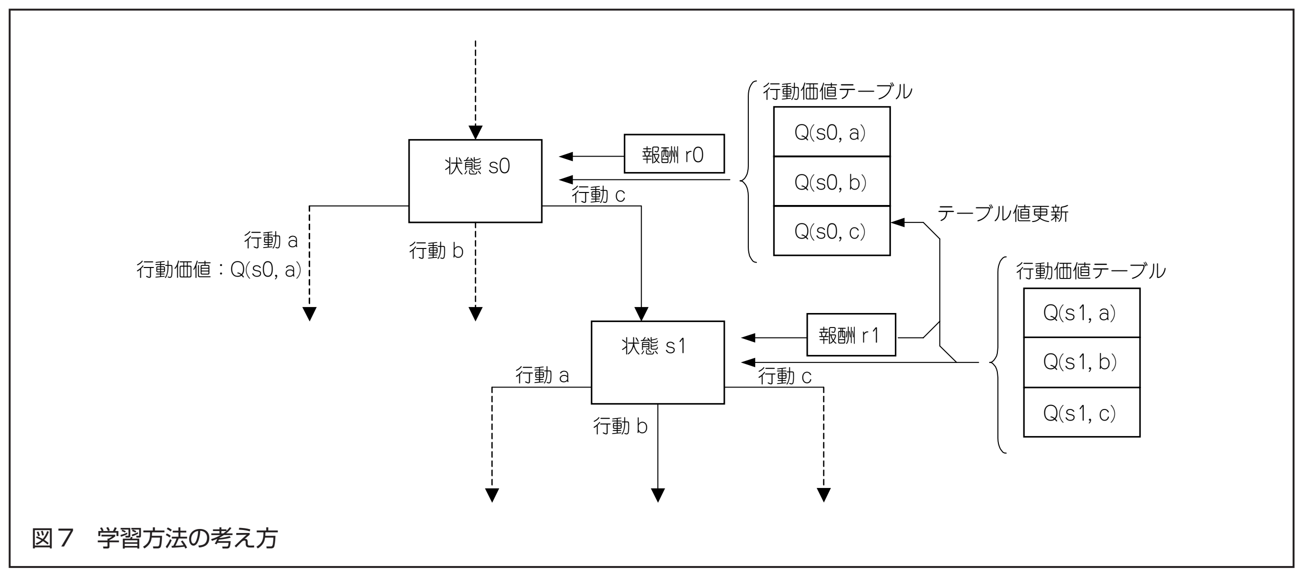

図７　学習方法の考え方

sarsaのアルゴリズム

　sarsaでは、s値とa値で決める行動価値テーブルの値(Q（s,a))を

1）前回のs値(s(t))

2）前回のa値(a(t))

そして、これらによって行動した結果としての

3）新しいr値(r(t+1))

4）新しいs値(s(t+1))

5）新しいs値から決まる次のa値(a(t+1))

の合計5つの要素を使って次のような演算処理で更新していきます。

$$Q(s(t),a(t))=(1-\alpha)\ Q(s(t),a(t))\ +\ \alpha[r(t+1)+\gamma Q(s(t+1),a(t+1))]$$

　ちなみにsarsaという名称はこの5要素の頭文字をとって並べたものです。要するに

・前回の行動価値に対して

・行動した結果受け取った報酬rと新しいsとaで決まる行動価値を少し足す

という操作で前回の行動の価値を上下させるというわけです。つまり、今回非常に高い報酬が得られたり、次の行動価値が非常に高くなるようであれば、前回の行動は正解であったということで、価値を上げておきます。逆に報酬が非常に低かったり、新しい行動価値が低いようなら、前回の選択は間違いであったと考え、価値を引き下げておくのです。

　これを図にしたのが**図7**です。線が入り組んでいますが、順にみていきましょう。状態s0から行動cを選択して状態s1になったとします。

　s0でもs1でも取りうる行動はa，b，cの三つ(今回の例なら右折・左折・直進と思えば良いでしょう)とします。s0から移動するときは行動価値テーブルの中からQ(s0，a)、Q(s0，b)、Q(s0，c）の三つを見て行動を決定したわけです。

　状態s0から行動cを終了してs1状態になったとき、報酬r1が受け取られました。ここで、s1からとりうる行動はやはりa,b,cの三つあります。状態s1からa,b,cのそれぞれの行動を行ったときの行動価値はQ(s1，a)、Q(s1，b)、Q(s1,c)に記録されているので、この中からひとつを選択します。図では行動bを選択したと考えましょう。

　ここで、状態s1で得られた報酬r1と、s1から新たに選ばれた行動bの行動価値Q(s1，b)を見てみます。もし、この両者が高い値を示していれば先ほど状態s0のときに行動cを選択したことは良いことであったと考えられます。そこでQ(s0，c)の行動価値を引き上げておくわけです。逆に報酬r1やQ(s1，b)が小さければこれは好ましくない選択であったということで、Q(s0, c)の価値を下げておくというわけです。

ε－greedy

Q(s(t), a(t))が選ばれたときはこの新しい行動価値データによって動きが決まるため、長い間このような試行錯誤を繰り返しながらテーブルを更新しているうちに次第により良い行動を選ぶような行動価値テーブルが出来上がっていくと考えられます。

強化学習では新しい行動a(t+1)の決定は大事な作業です。過去の経験の積み重ねである行動価値情報Q(s, a)の中からベストの選択、すなわちQ(s, a)が最大となるようなaの値を探して選ぶようにすることをGreedy(貪欲／食い意地の張った)な選択とも呼びます。過去の結果を利用(exploit)しているという状態です。一方、Greedyではない選択をすることは探査(explore)と呼びます。

過去の経験の利用ばかりでは、より良い結果が得られる可能性のある道を閉ざしてしまうことになります。一方、探査ばかりでは一向に良い選択をするようになりません。歴史や過去の経験に学ぶことなく、同じような失敗を何度でも繰り返してしまう状態と似ています。sarsaにおいては利用と探査の比率をどのようにするかがが大事な要素になってきます。

適切に比率を決めてやると、多少の間違いはしながらも次第に正解に近づいていき、動作中に環境条件が変化(今回の例ならば光源が勝手に移動したり、光源の明るさが変化するなど)しても新たな条件に対応した解答を見つけ出すことができます。

実際の実装では乱数を利用してある一定の確率で利用と探査のどちらにするのかを決めています。確率εでランダムに行動を決めて探査を行い、それ以外のときは最大の行動価値を持つQ(s, a)を選択するような方法をε－greedy選択と呼びます。

ソーラーカーにおける強化学習の実装

今回のソーラーカーではQ(s(t), a(t))を記録しているのが

```
char q_table[NR_STATE][NR_ACTION];
```

の2次元配列です。プログラムの詳細はソースコードを読んでください。肝心の強化学習による動作部分をmain()関数から切り出すと次のようになります。

```
while(1) {
    takeAction(action);                        // 行動a(t)によるモーター駆動状態の変更
    ・・・(中略：時間稼ぎのループ) ・・
    reward = senseReward();                    // 報酬r(t+1)：太陽電池出力電圧の変動
    next_state = senseState();                 // 状態s(t+1)：フォトセンサ出力変動取得
    next_action = selectAction(next_state);    // 次の行動a(t+1)の決定
                                               // sarsaによるQ(s,a)の更新
    updateQTable(state, action, reward, next_state, next_action);
    action = next_action;                      // 行動a(t)の更新
    state = next_state;                        // 状態s(t)の更新

    }
```

主要な関数の処理内容

次に簡単に各関数の動作を見ておきましょう。

- **takeAction(action)**

引数のaction（右折・左折・直進）によってモーターの制御ポートの状態を決定します。

- **senseReward()**

報酬rの値を得る関数です。今回はADチャンネル2につながった太陽電池の出力電圧の変動分を報酬としていま

す。プログラムは

```
AD_Start(ADChannel2);          // 現在値の取得
AD_Read(&ad_value);
if(ad_before_value == 0){      // 前回値がゼロ（初期値）ならゼロとする
        ad_before_value = ad_value;
        return 0;
}
else{                          // ゼロでないなら
                               // 前回値との差分から±50、またはゼロを選択
        int diff = ad_value - ad_before_value;
        ad_before_value = ad_value;
        if(diff > 200)  return 50;
        if(diff > 100)  return 0;
        else            return -50;
}
```

● **senseState();**　　// 状態s(t+1)：フォトセンサ出力変動取得

状態sの値を得るものです。今回はADチャンネル0につながっているフォトセンサの出力電圧の変動分を状態値としています。

```
AD_Start(ADChannel0);                  // 現在値の取得
AD_Read(&ad_value);
if(ad_before_value == 0){
        ad_before_value = ad_value;
        return 2;                      // 初期値は2にしておく
}
else{                                  // 初期状態以外なら差分に応じて0～5の値を返す
        int diff = ad_value - ad_before_value;
        ad_before_value = ad_value;
        if(diff > 200)   return 5;
        if(diff > 100)   return 4;
        if(diff > 0)     return 3;
        if(diff > -100)  return 2;
        if(diff > -200)  return 1;
        else  return 0;
}
```

● **selectAction(next_state)**

利用と探査の重みづけをして行動aを選択する、ε-greedyを実行する関数です。これは次のような処理を行います。

```
BYTE rand = random(20);
if(rand < 2){              // 探査の実行：確率0.1でランダムな行動aを選択
        return random(NR_ACTION);
}
else{                      // 利用の実行： q_table[state][i]が最大となるiを見つける
        max_q = q_table[state][0];
        for(i = 0; i < NR_ACTION; i++){
                if(max_q < q_table[state][i]){
```

```
                                        action = i;
                                        max_q = q_table[state][i];
                              }
                    }
          }
          return action;
```

今回は確率0.1(=2/20)でランダムな行動を、0.9(=18/20)でQ(s ,a)が最大となる行動を選択するようにしています。

random()は乱数を生成する関数です。擬似乱数を利用する方法もありますが、今回は未使用のアナログ入力ピンの状態が不安定になっているのを利用して、A/Dコンバータで未使用アナログ入力ピン(ADチャンネル1)を読み取った結果から生成しています。本格的に行うならば、ツェナーダイオードが生成するノイズから乱数を作るなど、物理乱数生成回路を付加したほうが良いでしょう。

● updateQTable(state, action, reward, next_state, next_action);

Q(s, a)をあらわすq_table[ ][ ]配列の値の更新です。引数はsarsaの並びそのものです。処理は次のような具合です。

```
#define alpha 0.9
#define alpha 2  0.1
#define beta  0.9
          int tmp   = alpha * q_table[s][a];
          int tmp2 = beta * q_table[ns][na];
          int tmp3 = r + tmp2;
          int tmp4 = alpha2 * tmp3;
          q_table[s][a] = tmp + tmp4;
```

変数tmp等を使っているので、少々わかりにくいかもしれませんが、これで先に示したsarsaアルゴリズムの式である

$$Q(s(t), a(t)) = (1-\alpha)\cdot Q(s(t), a(t)) + \alpha[r(t+1) + \gamma Q(s(t+1), a(t+1))]$$

で、α=0.1、γ=0.1とした演算処理を行っています。

まとめ

強化学習を使用した知的制御のサンプルとして、太陽電池の発電量が大きくなるように行動するソーラーカーをとりあげました。状態がフォトセンサ、報酬が太陽電池、行動が右左折直進という、比較的シンプルな入出力構成ではありますが、強化学習の考え方や学習の進行に伴う挙動の変化などを実際に目で見ることができるので、非常に興味深い製作事例と言えます。

知的制御や強化学習などという言葉を聞くとおおよそ理解しがたいほど難しそうな事に思えるかもしれません。もちろん、背景にある理論そのものは本が一冊書けるほど奥深いものではありますが、実際のプログラムに展開されてみれば、理解しがたいほど複雑怪奇なものでもないと思えるのではないでしょうか。

是非一度製作してプログラムにいろいろ手を加えるなど楽しんでみてください。

編集　桑野 雅彦