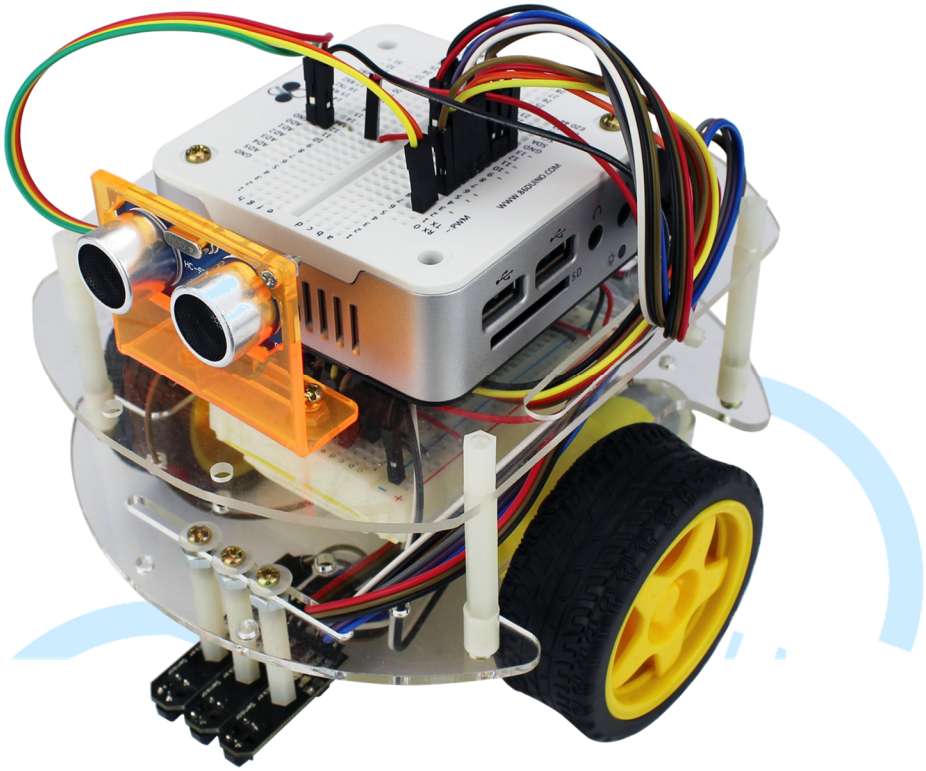
EduCake　實作自走車



一、　自走車該有哪些功能

大致介紹過 86Duino EduCake 各種訊號控制、互動的方法之後，這個章節我們就來實作一個實際的應用範例-自走車。自走車在很多地方都派得上用場，最大名鼎鼎的就是登陸火星，像普通的小轎車一樣大的好奇號，至今還在火星地表為人類探險和偵查火星的各種大地和地表資訊，不斷的傳回高解析火星地表照片，有興趣的讀者可以參閱維基網站 http://zh.wikipedia.org/wiki/%E5%A5%BD%E5%A5%87%E8%99%9F。登陸火星對你我來說是根本不太可能的事情，但眼光放到地球上，還是有不計其數的各種自走車應用在生活中，像是 GOOGLE 已經快接近實用階段的自動駕駛車，

都已經開在路上超過百萬公里，也已經開始市區道路的測試動作(http://www.ithome.com.tw/news/87223)，預計未來的五六年內就可以真正實用到普通車上；像是公司的門面用來接待客人、送茶水的機器人目前也很多單位在實作和使用；像是各種工廠內，需要搬運大型零組件的時候，都會使用自動搬運車來幫忙精準快速的把貨物搬到指定位置，提高生產效率；像是 Amazon 這種大型量販中心，為了快速組合出客人的訂單，都會有很多的自動運送自走車再幫忙在倉庫內四處奔走收集貨品；而生活中，最常見的自走車就是掃地機了，一般人若是要求不高，一千多新台幣就可以買一台來家裡幫忙打掃，而掃地機也有很高級一台三五萬的版本，擁有掃地乾淨、路徑規劃精準、自動回歸充電、定時清掃、不碰撞障礙迴避等等的功能。

這些不同種類的車都由各種不同動力、感測器組成主要功能，底下就依照功能分類來談談這些功能如何使用 86Duino EduCake 來實作一台簡單的自動搬運自走車。

二、 動力和運動控制

車子當然是要輪胎能動，才能帶動車子前進，動力部分，這只是一台示範性的車，不需要載重和特殊的功能，使用簡單的減速齒輪箱和普通的 DC 馬達構和 65mm 塑膠輪台構成的普通組合就行了，如附圖 1。這種馬達使用電壓限制為 0~6V，0V 時車子完全不會動，慢慢的增加電壓，約是到 1V 時馬達就會開始慢慢的旋轉，在繼續增加電壓，馬達就會慢慢的越轉越快，且因為需要的電流約是 100~500mA，兩顆最大約 1A，這使用普通的 L293D 雙馬達驅動 IC 就可以輕易控制了。

圖 1. 小型車用的減速馬達和輪胎

控制馬達轉動的電路如圖 2。因為需要有轉速的控制，所以每顆馬達共是兩條方向訊號線和一條 PWM 速度控制訊號線，兩顆馬達共需要六條線才能正確使用。圖中的 1.2.3 號線控制第一顆馬達，4.5.6 號線控制第二顆馬達，這六條線就是找 EduCake 上面尚未被其他裝置使用到的 digital 腳位來安裝。控制每顆馬達的轉速可使用 **analogWrite(EN,速度);** 這個指令，這部分因為前面在談 PWM 的章節已經有說明過了，這裡就跳過不再贅述。注意馬達的電壓是給最高 6V，所以電路圖畫 4 顆 1.5V 的電池串連當示意，實際使用，是用變壓器和 DC converter 來降壓到 6V 供給他使用；或是使用行動電源的話，那就是 5V 要升壓到 6V 給他。另外這個極限電壓沒有一定要 6V，可以稍微高一點，筆者測試到 7V 都還可以使用，但在更高就會嚴重影響馬達壽命了，盡量是不要。

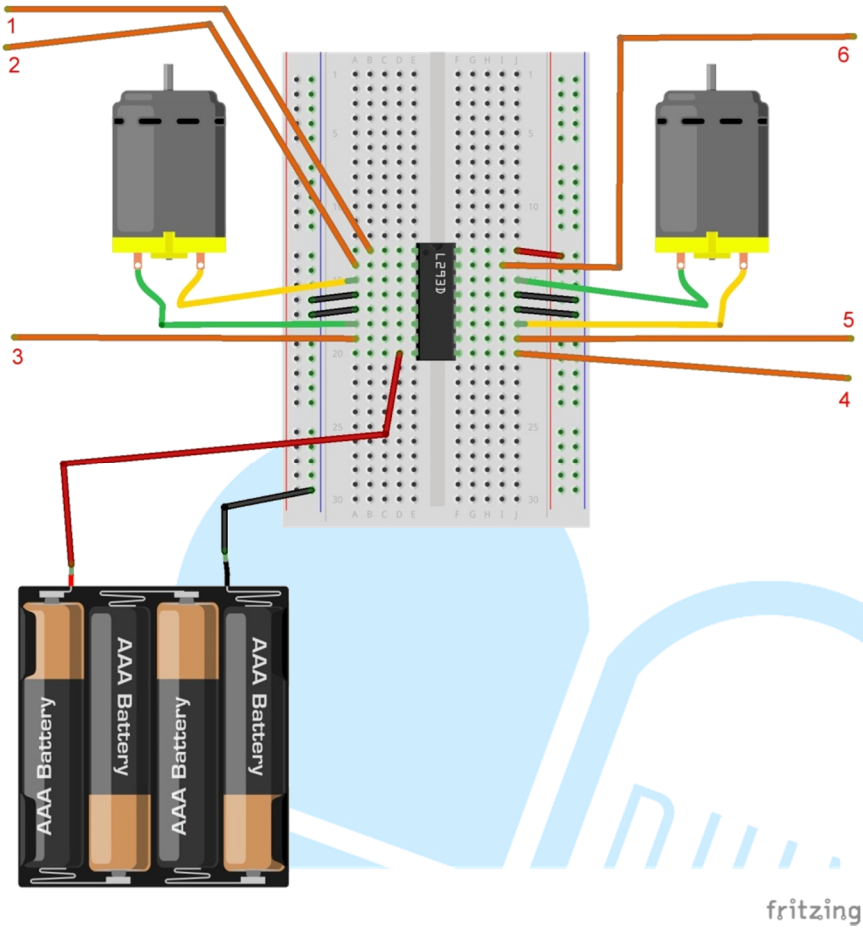


圖 2. 馬達控制電路

控制程式會長的像這樣:

```
int M1a=6,M1b=7; //     這組用來控制第一顆馬達正反轉
int ENa=5; //第一顆馬達的速度
int M2a=9,M2b=8; //     這組用來控制第 2 顆馬達
int ENb=10; //第二顆馬達的速度
void setup()
{
   pinMode(M1a,OUTPUT);
   pinMode(M1b,OUTPUT);

   pinMode(ENa,OUTPUT);
```

```
    pinMode(M2a,OUTPUT);
    pinMode(M2b,OUTPUT);
    pinMode(ENb,OUTPUT);
}
void loop()
{
  int a;
  for(a = 0; a <= 255; a+=5)
  {
    digitalWrite(M1a,HIGH);
    digitalWrite(M1b, LOW);
    analogWrite(ENa, a); //  使第一顆馬達由慢到快慢慢加速
    delay(20);
  }
  digitalWrite(ENa, LOW); //  停止
}
```

再來就是想要知道車子的動作狀況，需要使用編碼器/編碼盤(Encoder)這樣的裝備，編碼器如圖 3

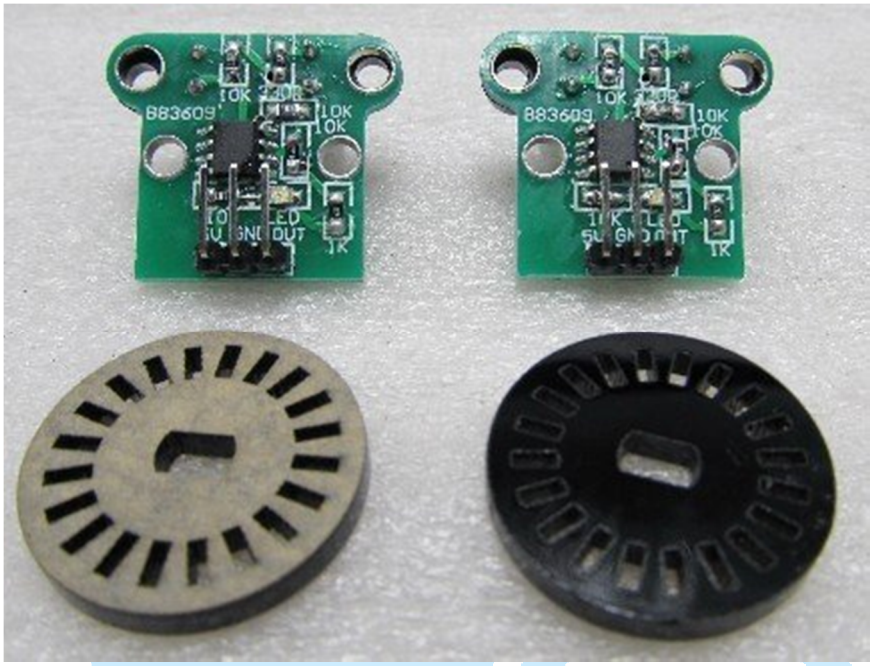


圖 3. 編碼器

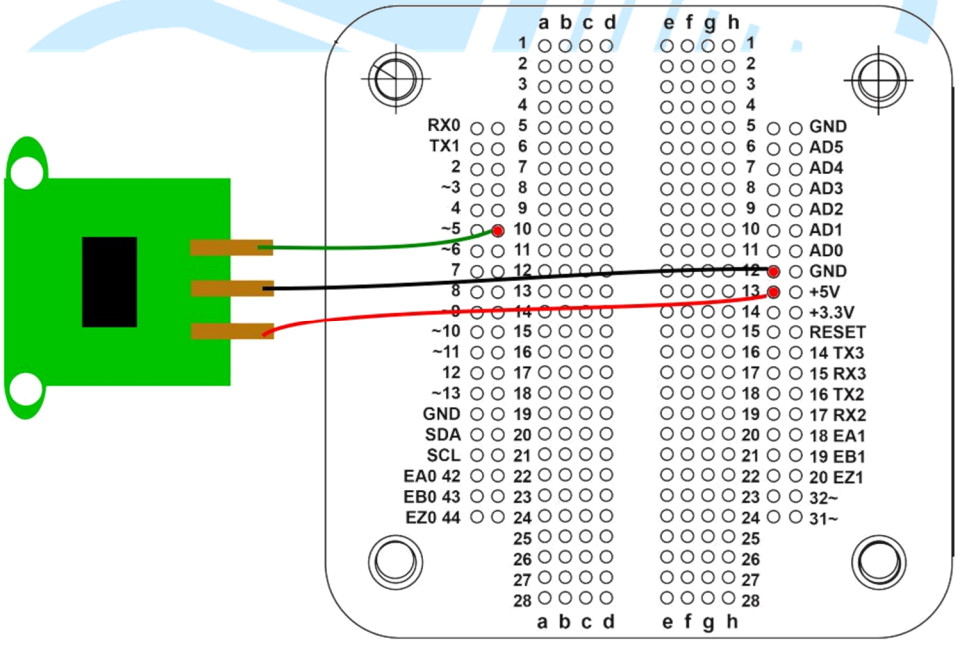


圖 4. 編碼器接線圖

編碼盤通常裝在輪子軸心上面，輪子轉動時，帶動編碼盤(畫面中一格一格那個就是)一起轉動，編碼盤會遮住光線，但上面有洞的地方可以讓光通過，利用光電感應器就可以直接計算取得到底轉了幾格，換算成輪子的轉動距離。

實際使用的時候，因為輪子轉動很快，這不能利用迴圈一直去作 digitalRead 讀取感測器動作，除了效率差而且很不準，一直 Read 也不一定能讀取到變化的位置，造成很大誤差，還會害程式卡在那個位置。簡單的作法應是使用 pulseIn(); 函數，因為編碼盤有洞的地方會造成光電感應器傳回 1(光有通過)，沒有洞的地方會傳回 0，這時候使用 pulseIn(pin, HIGH);可以量取到轉過一格有洞的位置需要多少時間，由編碼盤上面算得孔洞共有 20 格，沒有孔洞的位置也是 20 格，把量到的時間乘上 40 就可以大概知道轉一圈需要的時間，這就是轉速了；而因為輪子的周長是已知，所以轉動一格前進的距離也可以輕易的計算出來。舉例來說，輪子直徑是 6.5cm，則這顆輪子旋轉一圈能前進的距離就是：6.5*3.14=20.41cm(當然，這裡的圓周率使用越精確的數值，算出來的結果就越準，但得考慮選用的控制板的運算能力和能容納的精確度)。編碼盤被分割成有 20 孔洞，前面的數字為了計算方便概算 20cm 的話，則每一格就代表前進了 20cm/20 格=1cm。這樣的結果其實是有很大的誤差在的，除了圓周率的因素以外，車子可能在某些地方造成打滑、輪胎因為載重造成的變形等等都會使得量測結果和預料不一樣，一般都會再搭配額外的方式來協助修正誤差，像是檢查點、影像、或是測距修正等等。

前面這個方法有個大問題，車子萬一不動時，使用 pulseIn 函數有個問題，就是一樣會讓程式卡在那邊直到函數的 timeout 時間，timeout 的長短也不容易設定，因為我們不曉得車子會跑快還是跑慢(除非，我們一開始就指定車子只能用某些速度跑)，這樣很不理想，更好的方式是使用中斷+millis()函數，可以很精確的量取到每格經過的時間，這也是前面章節談過的，就請自行回去翻閱；另外把編碼盤的挖空格數增加也是一個好方法，等於縮短每個空格的距離，像是

由 20 格改成 100 格就能明顯變精確；還有一個方式是改變安裝位置，例如改裝到 DC 馬達的後方出軸，而不是現有裝到減速齒輪箱的輪軸位置，若以減速齒輪箱的減速比是 150:1 來說，輪軸位置轉動一圈等於後方 DC 馬達出軸已經轉動 150 圈，這可以直接提高 150 倍精密度，但相對這樣需要的處理速度也要更快，若使用中斷的解法會造成中斷太頻繁。中斷的觸發也是要一點 CPU 時間的，很頻繁的中斷觸發等於會消耗大量的 CPU 時間，萬一 EduCake 還需要有別的運算要作，這樣會影響效能，如何取捨還得實驗看看。中斷的程式碼如下:

```
volatile int counter ;
unsigned long oldtime;

void setup()
{
  pinMode(pin, OUTPUT);
  attachInterrupt(2, RPM, RISING);//使用訊號由 LOW 變 HIGH
的時候引發中斷
  Serial.begin(9600);
  oldtime= millis(); //先把目前時間記錄下來
  counter =0;
}

void loop()
{
   long rpm;
   if (counter   >= 20) //  前面講過一圈 20 格，count=20 代表轉
一圈了
{
```

```
        rpm = millis() - oldtime; //  先取得共經過多少時間了，單位千分之一秒
        rpm = 60 * 1000 / rpm; //  計算轉速，一分鐘 60 秒，一秒 1000ms
        oldtime = millis(); //重新開始計時
        counter   = 0; //  計數器順便歸零
        Serial.println(rpm, DEC);
    }
}

void RPM ()
{
 //  每次引發中斷都會來這裡，只作一件事情，把變數加一
 //  中斷裡面的事情盡量簡單，這樣才可以減輕中斷的處理負擔
 //  也不會造成外面部分的程式受到太大影響
 //  編碼器的訊號變化如圖 5
 //  引發中斷的位置在最右邊那個紅框的地方，訊號由低變高的時候觸發
 //  每一個這種位置都會進來這個函數讓 counter 加 1.
 counter ++;
}
```

一分鐘是 60 秒，一秒是 1000ms ，除以轉一圈所花的時間，就能計算出每分鐘的轉速，取得轉速以後就知道前進的距離，很多事情就可以用公式來計算取得了。不過這裡要注意是，裡面使用了函數 **millis()**，這個函數只能計算到千分之一秒，萬一輪子的轉動速度過快，這函數會很不精確，也可以考慮改用 micros() 函數，可以計算到百萬分之一秒的精確，只是我們這台示範車採用的是每分鐘轉速才 70 轉的減速馬達，也就是一秒一圈左右，以中斷來看，一秒觸發約 20 次，每次約 50ms 的時間，millis()已經很足夠了。

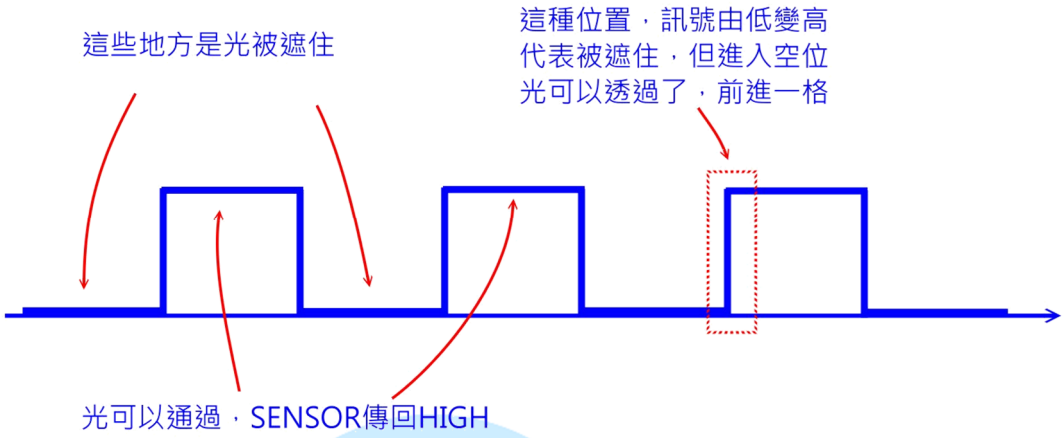


圖 5 編碼器的訊號變化

有了轉速，又知道車輪胎周長是65mm，那麼要計算車子一分鐘前進多遠、或是想要知道前進一公尺需要多少時間就都很好計算了，就可以用來作未來的車子行進策略。只是這種作法的準確度相對比較低，真的要高一點的準確度就得使用像是旋轉編碼器，解析度高一點的還得搭配運算速度比較快的 CPU 才有辦法作的比較好，成本也會明顯變高，這就後面再來談。

三、 置物箱開合

既然是自動搬運自走車，那就會需要有置物的貨架，這必須是一個密閉的箱子，想想看，若是想要把某個物品送到某人手上，那就不能讓自走車在途中被不相干的人拿走物品；製作便當配送車也是這樣，必須想辦法確保每個人都只能拿到一個便當，不然第一個人就把所有便當拿走，那後面的人都不用吃了。

要實踐這樣的概念，就要有個可以開/關的密閉箱子，這有很多種方式可以作到，電磁鐵、馬達等等，這裡選擇簡單的利用 SERVO 和簡單的機構來辦到，如圖 6，這是一個有蓋子可以打開的密閉盒子，和 SERVO 擺臂連接，就可以利用 SERVO 擺臂的上下來作盒子開盒的控制

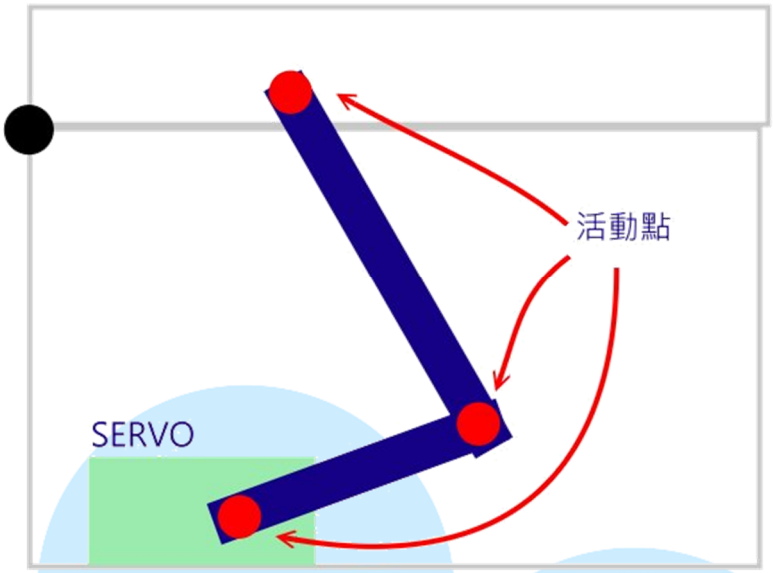


圖 6. SERVO 控制盒子

盒子原本是關閉狀態，這時只要 SERVO 的擺臂往上移動，就可以使盒子打開，如圖 7 這樣，SERVO 越往上推，蓋子就開的越高，這樣只要預先把盒子蓋上時的 SERVO 位置(假設是 PWM 1000us 的位置)和盒子打開到想要的地方的 SERVO 位置(假設是 PWM 1750us 的位置)都先測試好，那程式就可以簡單的控制盒子開關了。而至於什麼時後才讓 SERVO 作這個動作，則可以利用 RFID、遙控器、紅外線、藍芽...等等，有很多種解法，就看使用者如何選擇，筆者之前作辦公室便當自動配送車，是在每個員工的桌邊有一個 RFID，車子走經過這種 RFID 時就會停下來等五秒，每個員工手上的手錶背面也貼一個專屬該員工的 RFID，員工看到車子停下，可以用手錶去感應，車子就會打開，並推出一個便當，讓他拿走，這樣也可以順便控制一個人只能領一個不會多拿。當然該員工萬一當時沒有在車子旁那就不用領便當了，稍後再到車子停車區補領就好。

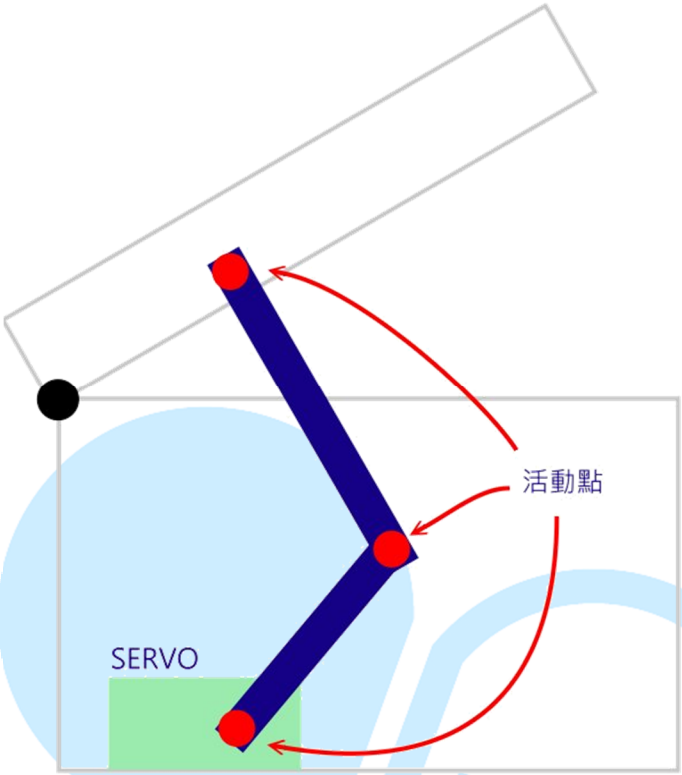


圖 7. SERVO 控制盒子的打開

寫好的範例程式碼會像是這樣:

```
#include <Servo.h>
Servo myservo; // 建立 Servo 物件，控制伺服馬達
void setup()
{
  pinMode(2,INPUT); // 設定由 digital 2 接一個按鈕
  myservo.attach(3);   // 連接 digital 腳位 3
}
void loop()
{
  int b=digitalRead(2);
  if (b==HIGH) // 代表按鈕被按下
    myservo. writeMicroseconds(1750); // 控制打開盒子
 else
    myservo. writeMicroseconds(1000); // 關閉盒子
}
```

這個程式就可以簡單的利用一個按鈕來測試，按下按鈕盒子會打開，放開按鈕盒子就會關閉，後面我們就可以利用這個功能來控制。這裡也闡述一個觀念，以前我們寫一個範例的時候都是以直接寫出來為原則，但後面因為漸漸會偏向比較大的程式專案，直接一口氣寫出所有東西，萬一最後專案不會動，會很難偵測出問題到底在哪裡，所以筆者通常喜歡這樣一個功能單獨一段寫出來，等最後功能都寫完了，再一段一段的拼起來，作出最後的成品，通常根據經驗這樣比較容易，遇到錯誤也好解決。這個功能也可以使用滑軌+步進馬達來作，效果會更好，但這只是台小車子，就不用作的那麼複雜了。

四、 紅外循線的應用

要讓自走車能夠精密的沿著既定的膠帶軌道行走，那就要讓自走車有感知的能力，使用影像辨識可以有最佳效果，但缺點是需要很強的 CPU 運算，攝影機取得影像也要夠快，這裡採用另外一種常見的簡單解法，利用紅外線尋線感測器，如圖 8

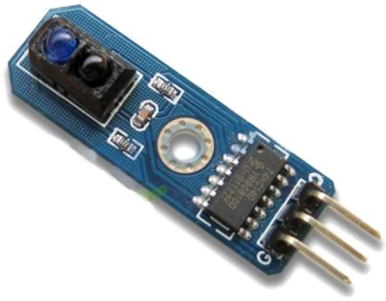


圖 8 紅外線尋線感測器

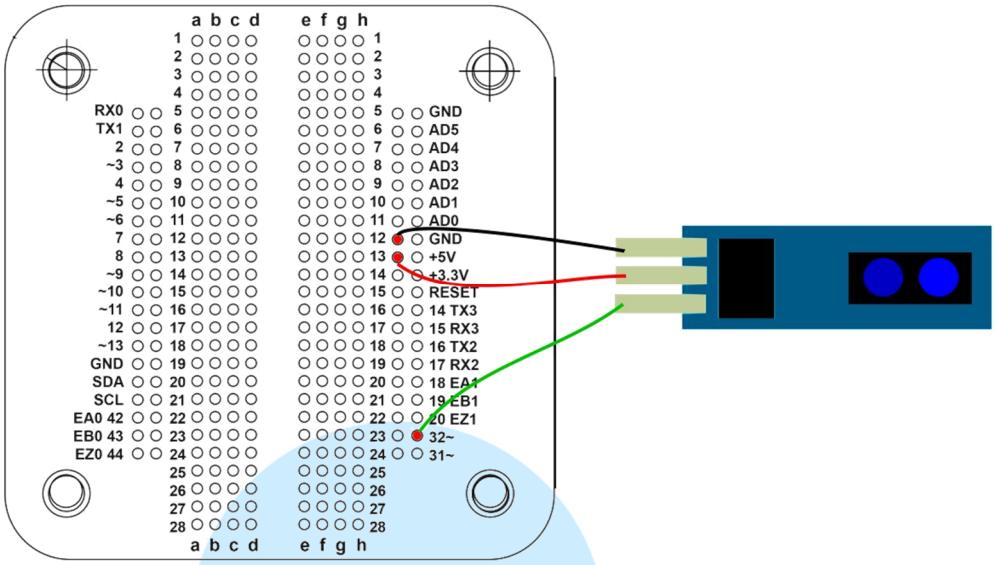


圖 9 紅外線尋線感測器接線圖

使用方式一樣是 GND、VCC 和一條接到 digital 腳位的 Sng。

這種感測器的原理一樣是利用紅外線，發射微弱的紅外線出去，由另外一個接收頭來感知反彈回來的光線，通常，多數顏色都能反射光回來，只有黑色會吸收大部分，就可以來判斷是否在黑色的線上面，目前多數的自走車比賽，使用的循線行走方式，都是透過這種感測器來實作的。使用時通常為了盡量的避免干擾，會把這種感測器盡量的安裝接近地面，約是 1~2mm 的位置。使用也很簡單，它的三條線分別是:VCC、GND、SNG。SNG 訊號線會在遇到黑線的時候輸出 0，其他多數顏色的時候輸出 1，就可以利用 digitalRead 指令來讀取感測器的狀態以便確認目前是不是在黑色膠帶上方。但這裡若是需要貼不同顏色膠帶來作額外的處理，那就需要使用顏色感測器了，作法會相對複雜很多，這裡先跳過不討論。

紅外線循線通常需要三個一組或是甚至五~七個，並排在一起放置，為什麼要這麼多呢？主要是因為看”想要克服的線”到底有多複雜而定。先來看圖 10

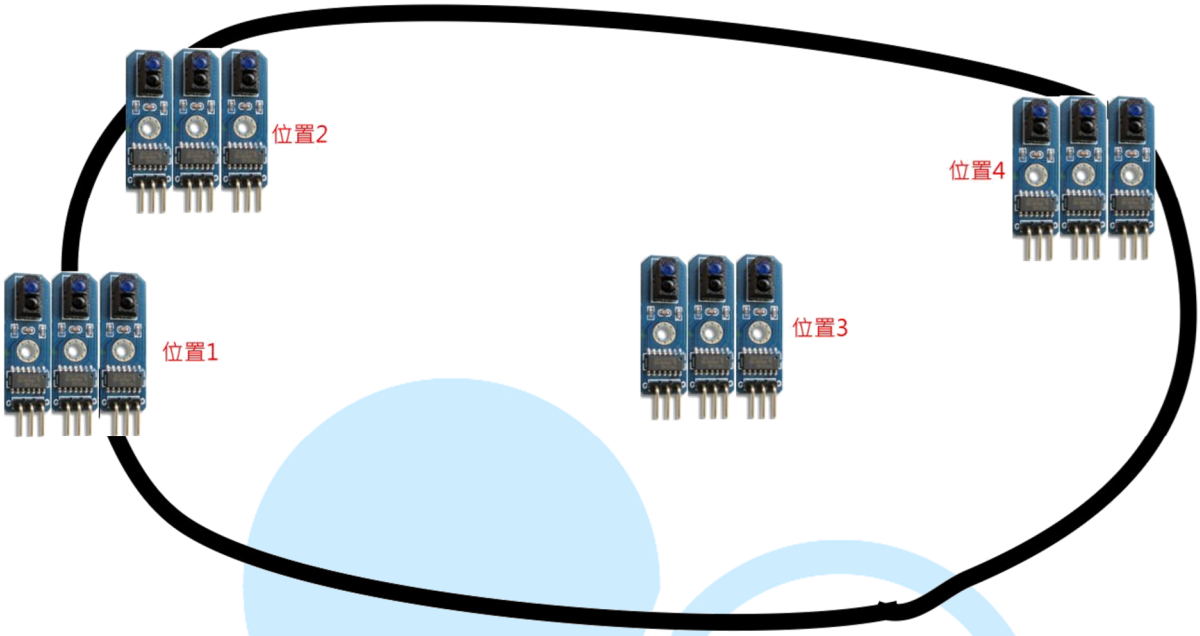


圖 10 紅外線尋線感測器遇到的各種狀態

這是一個簡單的地形，圖中使用了三個紅外線感測器並排在一起，用來偵測並沿著黑色的膠帶前進，只使用三個是因為場景只是個單純的一圈封閉圓，若是遇到複雜的形狀或是很多叉路的，那就要多裝幾個才能處理的了。筆者製作這個軌道的黑色膠帶(一定要黑色，這樣和地板的白色會有高度反差，辨識結果會很正確)是 1.5cm 的寬度，三個紅外線感測器彼此安排的距離是 1.2cm，所以若是把正中央的紅外線感測器對準膠帶的正中央時，中間的感測器會感應到黑色的膠帶，並傳回 digital　LOW 的訊號，左右兩邊則因為感應到地板白色的反光，會傳回 digital　HIGH 的訊號，程式就可以據此判斷，這時候車子正對膠帶，可以令左右的輪子同樣的速度向前轉動，直線前進，就像圖中的位置 1 那樣的情況。

再來看位置 2 的情況，車子的左邊和中間的紅外線感測器因為都壓在黑膠

帶上面，都會傳回 LOW 的訊號，但右邊的紅外線感測器在地板上傳回 HIGH 的訊號，程式可以判斷車子偏右了，導致同時兩個感測器都偵測到線，這時可令車子的右邊輪子轉快一點，左邊輪子轉慢點，讓車子偏左，回復到線中央。關於如何回到線中央，除了前面的講的兩個輪子轉不同速度(使用 analogWrite 就可以控制速度)來達成外，還可以直接讓兩顆輪子轉不同方向，以這個例子來說，可以使右邊輪子轉快，左邊輪子”反轉”，這樣的方向修正效果會很明顯比前一個快，主要是用在不同曲度的彎道，像是 90 度甚至更大角度的大轉彎，就要用這種方式才有辦法及時的過彎，甚至過彎前還得搭配兩個輪子同時減速，避免過彎太急而翻車。這在比較高速需求的電腦鼠領域裡面就會用上，得搭配更多個紅外線感測器萊偵測更大的範圍，取得較多的資訊作判斷。不過我們的車子行進速度並沒有那麼快，路況也只是大圓型沒那麼複雜，採用一種方式就好了。

位置 3，車子位於某處，紅外線感測器沒有偵測到任何東西，這時候有個問題，那就是車子到底在哪裡?有可能在整個黑色膠帶圈的範圍內，那就很容易，隨便往哪個方向開，都有機會直接遇到黑色膠帶，這時只要使用前面講過的方法就可以繼續使車子沿著線前進；但萬一車子這時是在線的外圍呢?! 要導引車子回到線上的難度就會比較高，不能使用隨便選個方向的方式，不然萬一選到背離線的方向，那車子就永遠回不來了。可使用在膠帶範圍中央放一個藍芽發射器，車子上也裝，這樣就可以利用藍芽的訊號強弱直接判斷出車子是遠離還是接近線圈，慢慢的修正位置回到線上，但這實作方式比較複雜容後再敘。

位置 4，這個狀況和位置 2 顛倒，正好是右邊和中間的感應器感應到黑線，前進時遇到這個情況代表車子偏左了，需要左邊的輪子轉快一點把車子帶回右邊；或是利用左邊輪子正轉，右邊輪子逆轉的方式，快速的修正方向回右邊。

總觀來說，以安裝三個感測器的情況下，共會有 2 的三次方等於 8 種狀況，

程式碼約是如下:

```
int Left_IR=5,Middle_IR=7,Right_IR=6;
int spd_a=255, spd_b=255; //  控制左右的速度
void setup()
{
  pinMode(Left_IR,INPUT);
  pinMode(Middle_IR,INPUT);
  pinMode(Right_IR,INPUT);
}
void loop()
{
  int L,M,R;
  int pos;
  L=digitalRead(Left_IR);
  M=digitalRead(Middle_IR);
  R=digitalRead(Right_IR);
  pos=L*4+M*2+R;
  switch (pos)
  {
  case 0://000   整個壓在線上，左轉(當然也可以使用右轉)
    turn_left();
    break;
  case 1://001，偏右一些，要左轉
    turn_left();
    break;
  case 2:// 010
  //左右有感應到線，中間沒有，這種情況在這個場景不會發生，先跳過不處理
  //但在有叉路的場景這可能代表正在叉路上，就不能跳過了
    break;
  case 3: //011，相對於 case1  ，這個情況偏右比較多，要轉多一點
```

```
      turn_left();
        turn_left(); //  連續執行兩次左轉
        // 或是另外用函數，或是用參數來處理旋轉時間，或是原地往左自轉
        break;
    case 4://100，  偏左一些，要右轉
        turn_right();
        break;
    case 5://101，在線正中央，直走
        move_forward()
        break;
    case 6://110，偏左很多，右轉多一點修正回來
        turn_right();
        turn_right();
        break;
    case 7://   111  沒感應到，直走，看能不能有機會遇到線
        move_forward();
        break;
    }
    delay(100);
}

void move_forward()
{
      digitalWrite(M1a,HIGH);
      digitalWrite(M1b, LOW);
      analogWrite(ENa, spd_a);   //左邊馬達速度
      digitalWrite(M2a,HIGH);
      digitalWrite(M2b, LOW);
      analogWrite(ENb, spd_b);   //右邊馬達速度
}

void turn_right()
{
```

```
  digitalWrite(M1a,HIGH);
    digitalWrite(M1b, LOW);
    analogWrite(ENa, spd_a);   //左邊馬達速度
    digitalWrite(M2a,HIGH);
    digitalWrite(M2b, LOW);
    analogWrite(ENb, spd_b-100);   //右邊馬達速度減慢，就有右轉效果
}

void turn_left()
{
    digitalWrite(M1a,HIGH);
    digitalWrite(M1b, LOW);
    analogWrite(ENa, spd_a-100);   //左邊馬達速度變慢就有左轉效果
    digitalWrite(M2a,HIGH);
    digitalWrite(M2b, LOW);
    analogWrite(ENb, spd_b);   //右邊馬達速度
}

void turn_left_rotate() //  向左邊原地旋轉
{
    digitalWrite(M1a, LOW); //  這兩行的狀態和 turn_left()顛倒，馬達反轉
    digitalWrite(M1b, HIGH);
    analogWrite(ENa, spd_a);   //左邊馬達速度
    digitalWrite(M2a,HIGH);
    digitalWrite(M2b, LOW);
    analogWrite(ENb, spd_b);   //右邊馬達速度
}

void move_back() //  後退，實際上就是前進的反方向旋轉
 {
```

```
        digitalWrite(M1a,LOW);
        digitalWrite(M1b, HIGH);
        analogWrite(ENa, spd_a);    //左邊馬達速度
        digitalWrite(M2a, LOW);
        digitalWrite(M2b, HIGH);
        analogWrite(ENb, spd_b);    //右邊馬達速度
    }
```

紅外線感測的另外一個用途是防摔落，有些時候可能自走車剛好走到樓梯邊，若是沒有圍住，車子可能就滾下去造成損壞了。一般這個問題有兩種解法，一個是在車頭使用紅外線對地面作測距，假設是一般的地面，測距傳回 10cm 就是正常，但走到樓梯邊，因為樓梯往下是忽然高度向下，測距會忽然傳回遠大於 10cm 的數字，那就要令車子立刻停下或是後退，如圖 11 這種 sharp 出品的紅外線測距感應器，可使用距離為 10 - 80 cm，用來量取是否到樓梯邊緣正適合。

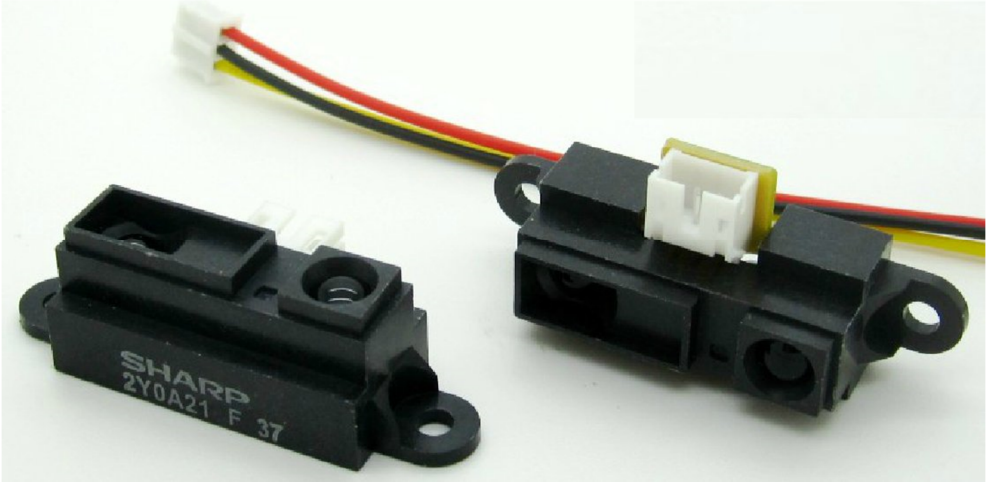


圖 11 紅外線測距感測器

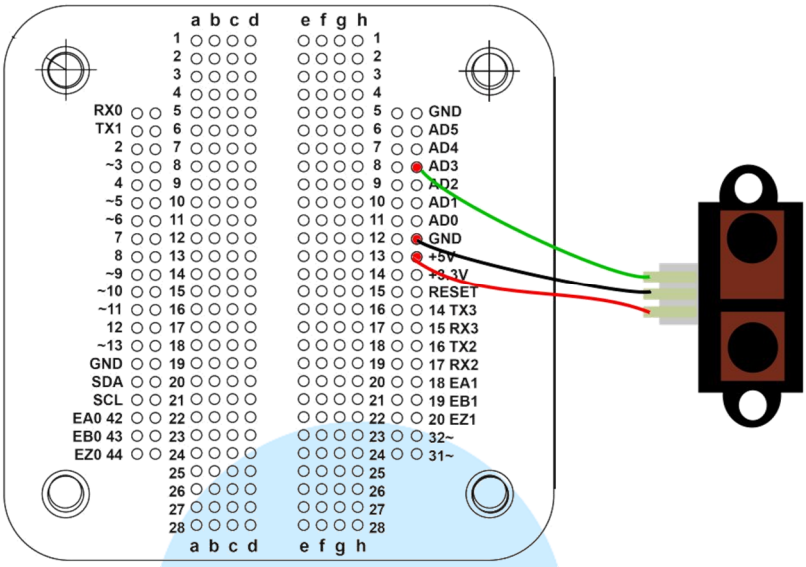


圖 12 紅外線測距感測器接線圖

控制紅外線測距讀取距離資訊的程式碼片段如下:

```
// GP2D12 紅外線距離感測
void setup()
{
  Serial.begin(9600);
}
void loop()
{
  int IR;
  IR = read_IR (3); //  接 analog P3  腳
  Serial.print("IR distance = ");
  Serial.print(IR);
  Serial.println(" cm");
  delay(500); //每隔 0.5 秒顯示一次距離
}

float   read_IR(byte pin)
{
  int tmp;
```

```
    tmp = analogRead(pin);
    if (tmp < 3) return -1; //  小於 3 不是正確數值
      return (6762.0 /((float)tmp - 9.0)) - 4.0;
    }
```

程式碼中的那段奇怪的公式(6762.0 /((float)tmp - 9.0)) - 4.0; 是因為傳回的電壓和距離成曲線比例關係，並不是直線的關係，所以，先取得約是 10~80cm 中每距離一公分的數值，填入 EXCEL 裡面，作線性化公式轉換以後得到的公式，這樣才可以比較直覺的取得紅外線測得的距離。

五、 超音波的應用

上面提到紅外線測距，對於距離的測量還算準確，但有個問題是測量的不夠遠，這時候就需要超音波的幫忙了。紅外線測距使用的原理是發出紅外線，照射到物體以後，利用感測器去檢測紅外線的回波，主要會遇到的干擾就是會發出熱源的物體在附近，會使得接到的回波受到干擾導致測得的距離不正確，但一般來說問題不大，最主要還是距離到一個程度，超音波過度的發散，導致接不到回波。超音波則是發出人耳聽不到的40KHz 的高頻音波，撞到物體後會反彈回來，接收器檢測到這個音波以後，就可以利用聲音在空氣中的傳播速度約是每秒 331 公尺(攝氏 25 度 C 時)，乘上感測器發出音波到收到音波的這段時間的一半，來計算距離，算是很直覺的一種方式

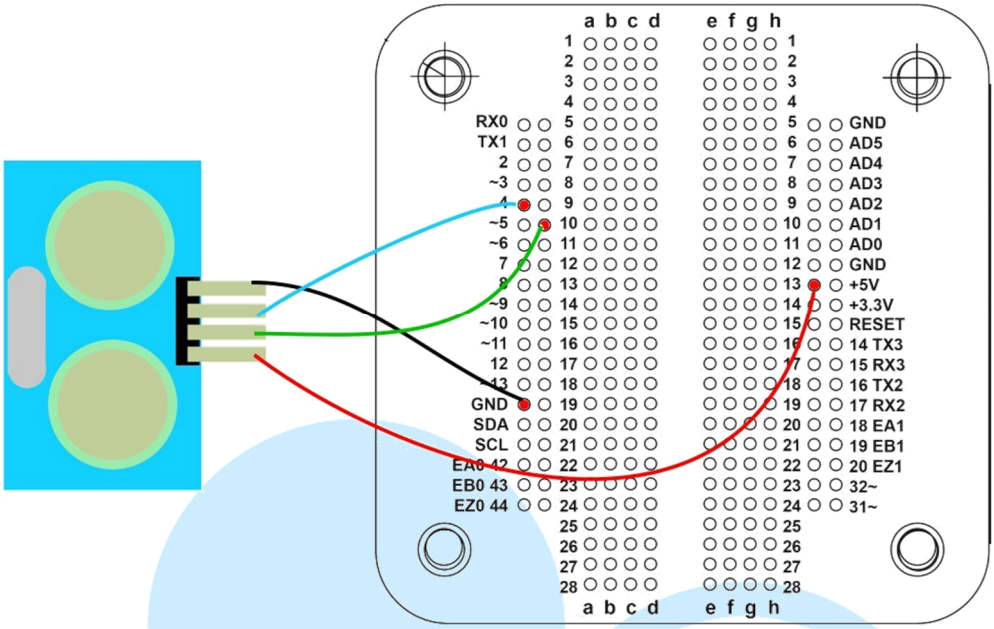


圖 13 超音波測距感測器接線圖

控制用的程式碼如下

```
int sonic_echo =4; //  接到 digital 4
int sonic_trig=5; //  接到 digital 5
void setup()
{
  Serial.begin(9600);
  pinMode(sonic_echo, INPUT);
  pinMode(sonic_trig, OUTPUT);
}
void loop()
{
    digitalWrite(sonic_trig, LOW);
    delayMicroseconds(2);
    digitalWrite(sonic_trig, HIGH); //  送出高電位 10μs
    delayMicroseconds(10); //  等待 10us  的時間，讓模組動作
準備好發出超音波
```

```
        digitalWrite(sonic_trig, LOW); //  時間到立刻設定為 LOW
          // 等待超音波回波讀取，並量取所經過的時間長短
          int distance = pulseIn(sonic_echo, HIGH);
          distance= distance/58;     // 轉換為 CM
          Serial.println(distance);
          delay(300);
        }
```

這個程式就可以簡單的讀取到超音波的距離，但其中的 int distance = pulseIn(sonic_echo, HIGH);取得的這個距離，約是號稱介於 1~500cm，為何說號稱呢? 因為超音波相比於紅外線的抗干擾能力更差，音波出去以後，有很多狀況導致答案其實不正確，像是遇到柔軟的海綿，可能完全被吸收沒有回波、距離過遠、沒有和被測物 90 度垂直，導致回波反彈到別處或是散射的過於微弱等等很多因素，都會害 pulseIn 函數發生 timeout，計算出來的數值當然就不正確了。筆者實際使用，約是 1~220cm 左右的範圍是可信的，超過範圍就開始很不穩定甚至完全量不到，測試原因主要是選用的超音波的散射角度太大，導致距離超過兩公尺以後就散的太大，根本抓不到回波，或說必須某個垂直正對障礙物的角度才能接到，但通常不可能剛好垂直，不過就算這樣也比紅外線好很多了，畢竟上面使用的紅外線也只有 80cm 的距離而已。

六、 掃地機路徑規劃簡介

接下來我們來談談掃地機路徑規劃的問題，以筆者在工廠碰過很多種掃地機的情況來說，一般掃地機的行進路線會想要盡量的涵蓋整個房間的所有地面，然後擴充到整個房子所有的房間。這樣的功能需求從簡單到複雜有很多種作法，

最常見就是沿著邊線走，然後撞到物體隨機改方向，這部分需要利用上極限開關來作碰撞感測(這樣才能做到輕輕碰撞的效果，總不能用力撞障礙吧)，把所有可以碰撞到物體的方向都安裝，一般來說是前方和側面，碰到物體時，極限開關被按下，程式就會知道這個方向有障礙，立刻改變馬達轉動的方向，然後繼續走，這種作法通常用在一千多台幣一台的入門掃地機，通常掃不乾淨，也掃不完全，這種作法我們只要用前面談過的功能來組合就可以輕易作出來

一般想要完全涵蓋整個室內面積，之字形、回字型兩種基本路徑規劃是常見的，底下就來看看這些作法。首先，要先了解一件事情，以掃地機來說，通長的處理方式是以"房間" 為單位，掃完一間在掃另外一間，或是說另外一區，而怎麼知道這是另外一區?通常得要車子有配備編碼器，並且走過整個房子內部，已經建立好房子的整個室內地圖，這時候就可以運算出每一個區域。

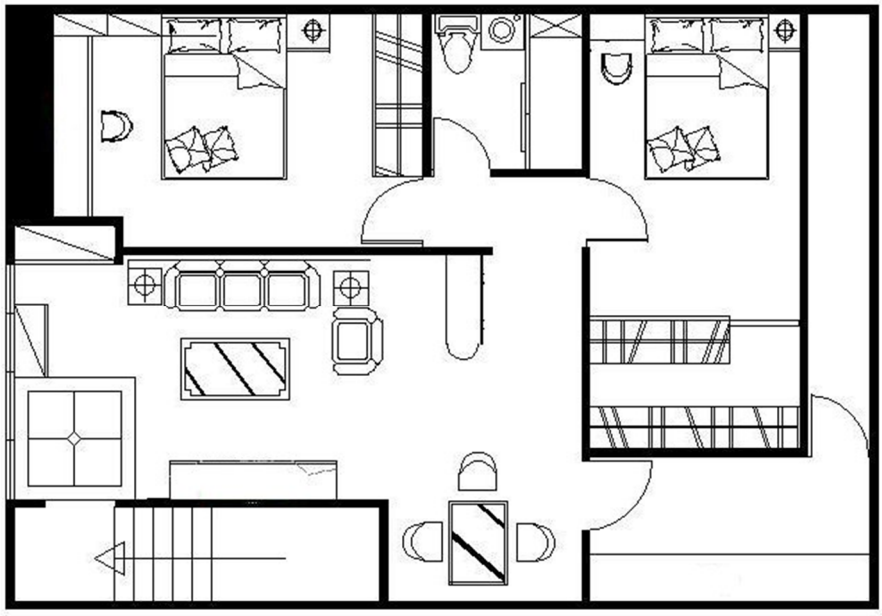


圖 14 一個正常的室內的地圖

圖 14 是一個房子內部的平面圖，車子先走過整個室內，因為車上的編碼器，車子走的時候可以對每個經過的位置作記錄，每個位置在車子的記憶體理模擬出來的大地圖上都是一個座標，每個座標可以簡單的分成可通過和不可通過兩種情況，等車子走過整個室內以後，一個虛擬的地圖就能在車子的記憶體裡面建構完成，這個動作叫做 SLAM(Simultaneous localization and mapping) 同步建圖與定位演算法，這實作的難度超過這個章節過多，容後再敘，這裡先簡述大致的理論就好。

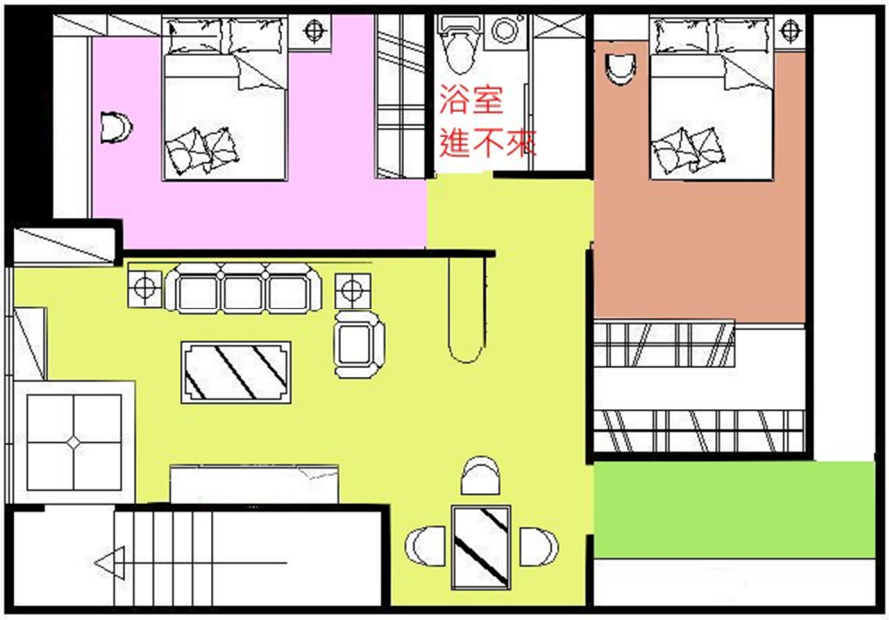


圖 15 處理分割後的室內的地圖

圖 15 是經過 SLAM 處理，並分割計算後得出的區塊圖，用顏色來看，可以看到分成左右兩間臥室，和左下的客廳和右下的廚房。主要的分割依據是門框。有了這樣的虛擬地圖和邊界後，就可以開始來作清掃的動作，先用簡單的圖示來看之字形和回字型行走法是怎麼回事，參考圖 16

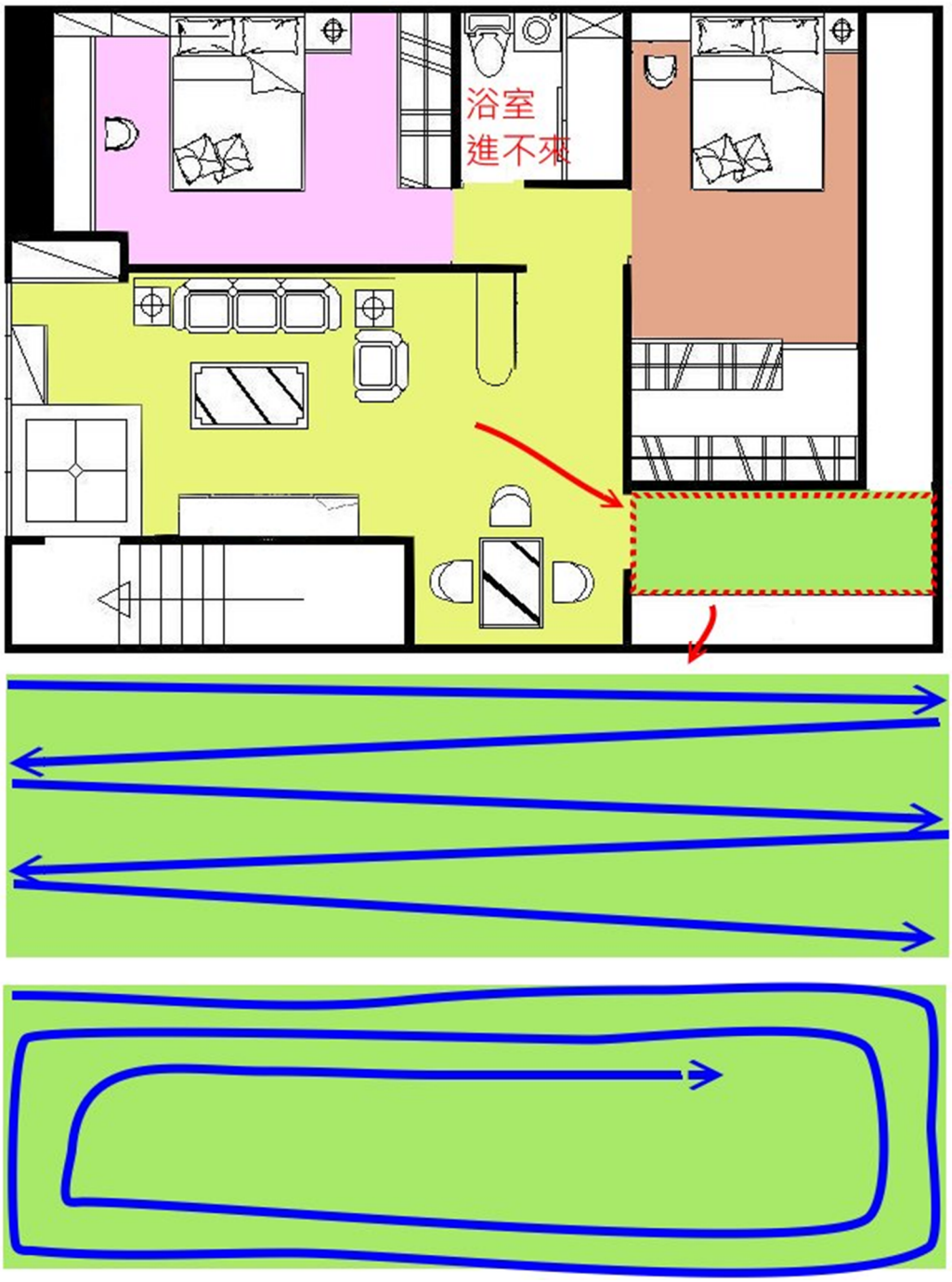


圖 16 針對廚房的兩種不同走法

可以看到圖中的之字形就是左右來回走，把整個區域覆蓋；回字型走法就是類似一直繞著區域邊緣，一直往中心方向走，要作到這樣的方式，就得善用感

測器取得的資訊，像之字形，就是一直往前走，直到撞到牆壁(撞牆偵測可以利用紅外線，或是掃地機常用的極限開關)或是遇到虛擬地圖中的邊界，原地旋轉 175 度(180 度就往回走了)繼續往前走到撞牆或虛擬邊界，這種原地旋轉 175 度的作法比起旋轉 90 度走一點點在旋轉 90 度的方式，容易造成有些地方會掃不到，但無所謂，通常掃地機為了掃乾淨會跑兩次，已基本能覆蓋到所有範圍，而且這裡只是為了簡單的講解原理，實作上還會因為硬體的區別或是環境的因素，有很多不同的細部調整，並不是很容易能辦到的。

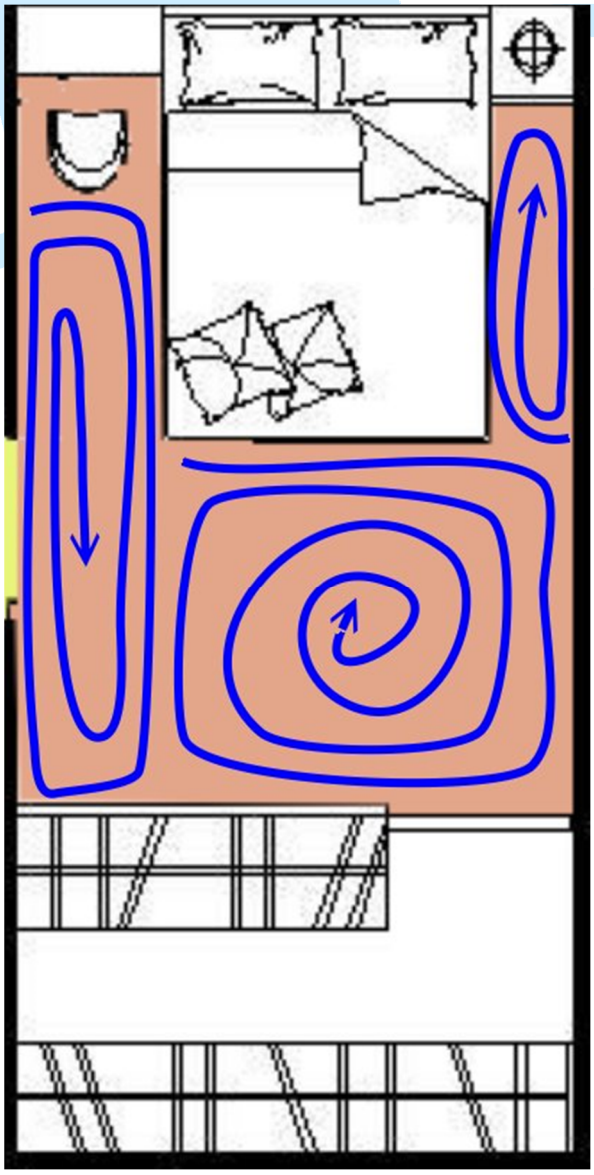


圖 17 更複雜的環境的區域覆蓋

七、 結論

自走車領域有很多的應用，而要達成這些應用需要很多各式各樣的功能結合在一起才能辦到。生活中的自走車也會因為各種需求的產生而越來越多，如何作出一台更實用的自走車還要靠讀者的想像力去發揮和實作，我們後面的章節還會談到更多的實作技巧來達成這些功能。