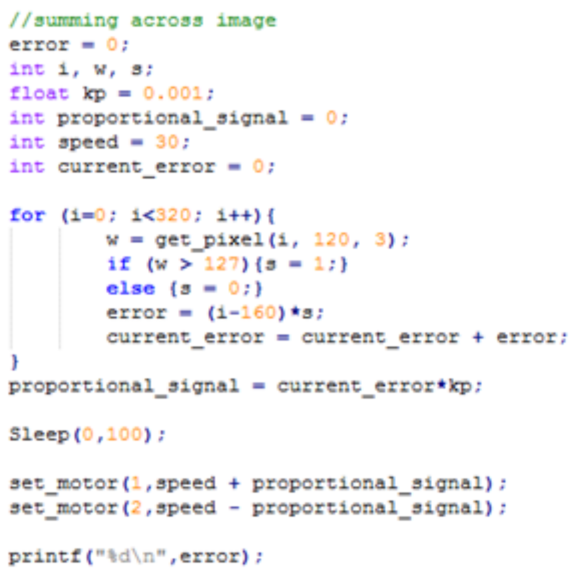
The following is Daniel's documentation up to Friday the 20th May:

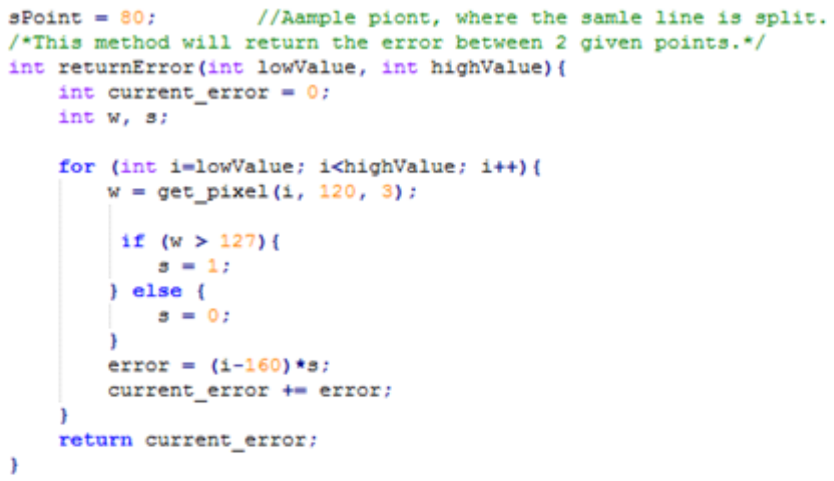
First version of code:



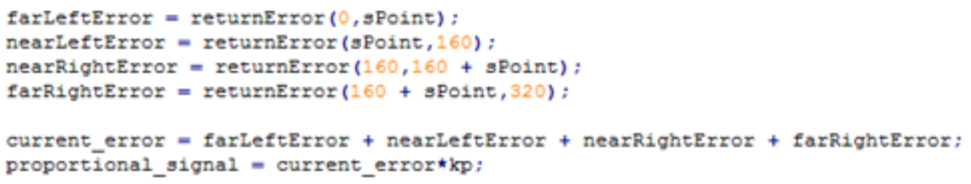
```
//summing across image
error = 0;
int i, w, s;
float kp = 0.001;
int proportional_signal = 0;
int speed = 30;
int current_error = 0;

for (i=0; i<320; i++){
        w = get_pixel(i, 120, 3);
        if (w > 127){s = 1;}
        else {s = 0;}
        error = (i-160)*s;
        current_error = current_error + error;
}
proportional_signal = current_error*kp;

Sleep(0,100);

set_motor(1,speed + proportional_signal);
set_motor(2,speed - proportional_signal);

printf("%d\n",error);
```

The code above is the first set of code that we tested. It has implemented the first stage of PID control (proportional). The proportional signal equals the current error (sum) multiplied by the constant Kp. The proportional signal Is then entered directly into the set_motor line of code. After working out the bugs and running the code above, the robot can drive in a straight line and also follow the line around gentle curves. This code only scans one line of pixels across the image. In order to get more information from the image, our next step was to split the line into 4 segments so the robot could make decisions based on the error value of each segment of the line.

Second version of Code:



```
sPoint = 80;        //Aample piont, where the samle line is split.
/*This method will return the error between 2 given points.*/
int returnError(int lowValue, int highValue){
    int current_error = 0;
    int w, s;

    for (int i=lowValue; i<highValue; i++){
        w = get_pixel(i, 120, 3);

        if (w > 127){
            s = 1;
        } else {
            s = 0;
        }
        error = (i-160)*s;
        current_error += error;
    }
    return current_error;
}
```



```
farLeftError = returnError(0,sPoint);
nearLeftError = returnError(sPoint,160);
nearRightError = returnError(160,160 + sPoint);
farRightError = returnError(160 + sPoint,320);

current_error = farLeftError + nearLeftError + nearRightError + farRightError;
proportional_signal = current_error*kp;
```

In the code above, the current error has been split into 4 separate error values: farLeftError, nearLeftError, nearRightError, farRightError. This splits the image into 4 quarters and calculates error values for each quarter. The returnError(int lowValue, int highValue) method returns the error value for each quarter. lowValue and highValue define the edges of each quarter of the image. The current_error has now been updated to be the sum of all four error values.

Third version of Code:
Instead of scanning one line of pixels across the image, we decided to scan 3 lines horizontally across the image. Each line was scanned in 4 segments and an error sum calculated and stored was stored in a 2D array.

We found a problem where all the pixels on the right side of the camera returned weird values. After checking the code we knew it was a hardware problem. This problem was caused by the camera being not plugged in properly. This was fixed by switching off the robot and pushing the camera in. This fixed the problem.

Fourth version of Code:
On Friday 20, I changed the code a lot. Dan had the idea to read vertically as well as horizontally, so I have created 7 new methods. A checkAcross method which scans across the image at a given height and returns true if there are more than 10 white pixels. There is also a checkDown which does the the same as checkAsross but checks a vertical line at a given position.
The other 5 methods are; top, mid, bot, left and right which calls one of the check methods. For example, top calls return checkAcross(60); which will scan across the image at height 60 and pass on the boolean value. This means that I can easily call call top() or !top() when checking what is in front of the robot. This picture shows what lines should be true or false for each different scenario.

T= true
F= false
X = Not needed / don't use

Left
Top
Mid
Bot
Right