In this example we’ll build a simple motorised gimbal. Gimbals are an ancient device used to stabilise, for example, compasses on board ships. In short they are designed so that when the ship moves, the compass stays flat.

Recently, with the introduction of small motors and small cameras, motorised gimbals have been used in photography – both for handheld cameras and for those mounted on, say, helicopters or UAVs. In these, a sensor detects changes movement of the person/helicopter and causes the motor to move the camera to compensate. And so the picture stays steady.

We will use the accelerometer to detect where down is and a servo motor to move the Engduino so that it stays in the same orientation. To do this, we’ll have to learn a little control theory – and you will undoubtedly find out that it requires a bit of time and effort to make a good(ish) control system.

To build this project we need to do two things: (i) figure out which direction is down relative to how the Engduino is being held; and (ii) move the servo motor to compensate. The second is itself two parts: (iia) learn how to move the servo motor to a given place; (iib) choose how much to move it in one go.

WHICH WAY IS DOWN?

You should complete the accelerometer handout before proceeding.

The accelerometer returns three values – x, y, and z. If we’re not accelerating the device, then the values are due exclusively to gravity and we can figure out what direction gravity is in just from the values returned.

First, we need to know what direction is what. Look at the x, y, z values as you move the Engduino in space. Check that the directions on the picture match the direction you think gives a positive value for each axis.
So, let’s make the problem a bit easier. If we hold the Engduino vertically (so the E always faces out from your chest), then the z axis will never have any acceleration due to gravity, irrespective of how the Engduino is rotated. So we only need to think about the x and y axes.

There is a useful function for working out angles from x and y values. The function is an arctangent — the opposite of a tangent.

\[
\tan \theta = \frac{y}{x} \\
\theta = \arctan \left( \frac{y}{x} \right)
\]

In Arduino code, the arctan function is represented by something called `atan2(y,x)` (and it is important to remember the 2): it’s a slightly more sophisticated form of arctan that gives a value between -180° and +180°

You need to add:
```
#include <math.h>
```
to the top part of the file with the other includes, and then you can use the `atan2` function:

```
angle_in_radians = atan2(accelerations[1], accelerations[0]);
```

Amend the code you wrote for the accelerometer to print out this angle, then rotate the Engduino (remembering to hold it so the E always faces out from your chest) so the angle changes.

The number seems to go from about -3.14 to +3.14 – what’s happening? Well, this is an angle in radians, and that’s perhaps something you aren’t used to, but it is no big deal — it is just another way of writing angles that happens to be useful to mathematicians. There are 2π radians in a circle — put another way 2π radians is the same as 360° or π radians is the same as 180°. So when we see numbers that go from -3.14 (i.e. –π) to 3.14 (i.e. +π), what we’re actually seeing is the number going from -180° to +180°. If you want to convert between an angle in radians and one in degrees we write the following code:

```
angle_in_degrees = angle_in_radians * 180/PI;
```

Note the capital letters for PI. Try it.

So, with the E the right way up and out from your chest, what angle would you expect? (Answer -90, but why?)
Next we’re going to take a look at the servo motor. These are used in radio controlled vehicles and aeroplanes to control, for example, the angle of the rudder of a boat, or the ailerons on a plane. They are a motor we can set to a certain angle – in this case ranging from 0° to 180° – only half a circle.

The motors are set using what is called a pulse width modulation signal (PWM).

### WRITING OUR OWN CODE TO DRIVE THE MOTORS

We can write a program to generate these by hand. Get the assistant to connect the device up for you. The servo motor control signal is connected to digital pin 5 on the Engduino.

```cpp
int motor = 5;                   // Motor connected to digital pin 5

void setup()                     // sets the digital pin as output
{
    pinMode(motor, OUTPUT);
}

void loop()
{
    digitalWrite(motor, HIGH);      // sets the LED on
    delayMicroseconds(1000);        // waits for 1ms
    digitalWrite(motor, LOW);       // sets the LED off
    delayMicroseconds(19000);       // waits for 19ms
}
```
This generates the first square wave (on for 1000μs = 1ms off for 19000μs = 19ms). It won’t quite work to set it to location zero because there are some extra delays in the instructions that make the on-time a little more than 1ms.

Experiment – see what works.

AN ALTERNATIVE

Fortunately, someone has done the hard work and written a library for us that will move the servo motor. The library the pulse width modulation (PWM) that is built into the microprocessor.

To use the library we have to ‘attach’ ourselves to the servo in the setup() routine and then we can command it to go to a particular angle by ‘write’ing a position in degrees. The code below is an example from the authors of the library – it can be found at http://arduino.cc/en/Tutorial/Sweep to save you retyping it (but you must remember to change the pin to 5). See what it does.

```
#include <Servo.h>

Servo myservo;  // create servo object to control a servo
                 // a maximum of eight servo objects can be created

int pos = 0;    // variable to store the servo position

void setup()
{
    myservo.attach(5);  // attaches the servo on pin 5 to the servo object
}

void loop()
{
    for (pos = 0; pos < 180; pos += 1)  // goes from 0 degrees to 180 degrees
    {
        myservo.write(pos);              // tell servo to go to position in pos
        delay(15);                       // waits 15ms for the servo to move
    }

    for (pos = 180; pos>=1; pos-=1)    // goes from 180 degrees to 0 degrees
    {
        myservo.write(pos);              // tell servo to go to position in pos
        delay(15);                       // waits 15ms for the servo to move
    }
}
```

So now we can set the servo to a known angle. In reality, we don’t necessarily quite get the full 0-180 degree range, but we can let that go.
MAKING A GIMBAL

Ask the class leader to attach the servo to the Engduino if it is not already attached. This just needs to be done so that the Engduino can move in the right way (set the servo to angle zero and then attach the servo with the top facing the same direction as the E).

Our goal is to create a system that ensures that the Engduino is always facing the right way up, no matter how we move it (with the bounds of what the motor can do). We know ‘the right way up’ is to have an angle of -90°. In control terms, this is called the ‘setpoint’.

If the angle changes because we move the Engduino, then we need to make the servo move in such a way as to compensate for that change. We do that by calculating an error between the setpoint and the current angle.

\[
\text{double error} = \text{setpoint} - \text{angle};
\]

ASIDE: CONTROL

If we have an error, then know exactly how far we need to move the servo to get to the correct position, right? All we would have to do is to write some code in the setup routine that says:

```
myservo.attach(5);  // attaches the servo on pin 5 to the servo object
myservo.write(pos);
```

And in the loop that says:

```
pos += error;
pos = constrain(pos, 0, 180);
myservo.write(pos);
delay(50);
```

The call to ‘constrain’ just ensures that the value always lies between 0 and 180 – the only sensible range of inputs to our servo.

Try it – see what happens, but be prepared to hang on. Unless you’re supremely lucky the system will be unstable. Let’s try something different - so called proportional control. What we do is to multiply the error by a number less than 1 (i.e. a proportion) – say 0.1. Replace the first line above by:

```
double Kp = 0.1;
pos += error * Kp;
```

Is it more stable? How about for different values of Kp, say 0.3? Keeping it in the same orientation, just shake it slightly and see what happens. In general, the higher the value of Kp, the more quickly it will get to the setpoint, but the less stable it will be.

What about more sophisticated control?

Well, a common form of controller is a PID controller – proportional, integral, differential.

- **Proportional control** is what we’ve just done – it depends on a proportion of the current error.
- **Integral control** adds up error over time. The bigger the accumulated error, the bigger the control input (e.g. the more off course a ship has become in spite of proportional control, the more sharply we should steer)

- **Differential control** depends on the rate of change. If we get a sudden change in something then we guess that that change will persist into the future – it’s a guess about what might happen.

Pseudocode would look something like the following:

```plaintext
error = setpoint - sensor_value;
errorSum    += (error * timeChange);
errorChange = (error - lastErr) / timeChange;

p = K_p * error;
i = K_i * errorSum;
d = K_d * errorChange;

pos = pos + p + i + d;
lastErr = error;
```

You could code this up and try some parameters. What you are likely to find is that many parameters do not lead to sensible behaviour, and it takes a little while to get a set that is much good – reasonably stable and with a reasonably rapid response. Fortunately, in real life there is maths to help with all of this – and some tricks on implementation that make the results better… welcome to Engineering.

Control systems are used in all factories, cars, homes, in the production of electricity and the pumping of water, in rockets and in robots for surgery. Building them is a many many billion dollar business and there’s quite a lot to learn.