

RASPBERRY PI: BUILD A MARS ROVER

BEN EVERARD

Polish your CV and call NASA: you're about to become a professional-grade robot builder.

WHY DO THIS?

- Get started with robotics
- Learn more about the Raspberry Pi
- Build a robot army and take over the world

“**R**obotics is a complex area that requires a combination of electronics understanding and the ability to use specialised machinery”. That last sentence is a common sentiment, but it's utter balderdash. Modern development boards like the Raspberry Pi (and the host of expansions that do with them) combined with the flexibility of Linux makes robotics incredibly easy.

To prove this, we're going to build a Mars rover-type buggy based on a Raspberry Pi. You'll be able to control it remotely, and it'll stream video back to the controller. To make control really easy, we'll build a smartphone app to use the phone's accelerometer, so you can drive the buggy by turning the phone (much like the controls in many smartphone video games).

There are quite a few parts to this, and we'll be using a few different technologies to control different parts, but thanks to the wide range of development tools on Linux, it's not as difficult as it sounds. For the hardware you'll need:

- Raspberry Pi and SD card (it is possible with a model A, but a model B will be easier to develop on).
- Raspberry Pi camera module (the NoIR module will be able to see in the dark).
- Raspberry Pi-compatible Wi-Fi dongle (see http://elinux.org/RPi_USB_Wi-Fi_Adapters).
- Power supply for Raspberry Pi.
- Power supply for motors.
- Two motors and drive train.
- One or two more wheels.
- Motor controller.
- Chassis.

You'll also need a Linux machine to do some development on, and an Android phone (other smart

phones should work, though you'll need the appropriate development environment).

If you haven't worked with robotics before, the final four might sound a little complex, but don't worry, they needn't be. While you could use almost any motors you can get your hands on, there are some easy, reasonably priced ones that are particularly easy to use from PiBorg (<http://piborg.org/accessories/dc-motor-gearbox-wheel>) and other suppliers. You only need two of these to drive the robot, and the only assembly is pushing the wheel onto the axle.

Reliant Mars Robin

For the final wheel (ours has three, but yours could have four), we used a ball caster (like this one: <http://shop.pimoroni.com/products/pololu-ball-caster-with-3-4-metal-ball>). This allowed the back of the buggy to move freely and follow the front two wheels.

The Raspberry Pi does have General Purpose Input and Output (GPIO) pins that can be used to switch low-power components like LEDs on and off. However, motors draw a much higher current than the GPIO pins can provide. Therefore you need some way of taking a signal from the Pi and converting it into an electrical current powerful enough to drive a motor. For the purposes of this project, we can classify these into two types: on/off controllers and variable speed controllers. The first (such as the PicoBorg or the relays on a PiFace) will work, but the controls won't be as finely-grained as they could be. We used a PicoBorg Reverse (<http://piborg.org/picoborgrev>), which enables us to vary the speed of each motor (other controllers are available with the same features). The most important thing is that the board you use as the brains of the robot should be controllable from Python (almost all are). There should be sample code on the board's website to show you how to do this.

The build

The chassis can be as simple or as complex as you like. Specialised robot chassis are available that are robust and capable of carrying lots of sensors. We don't need this much for a simple buggy though. You can use anything provided you can mount the wheels on it and it will support the electronics.

Finally, we used a USB power pack and a 9V battery to power the Pi and the motors respectively. This is quite a lot of hardware, but all of it could be used on other projects.



An ice cream tub makes a simple and cheap robot chassis – just make sure you wash it out first.

Obviously the build will vary depending on exactly what parts you've chosen. For us, it involved connecting the PicoBorg Reverse according to the instructions on the website (<http://piborg.org/picoborgrev/install>).

To set up the buggy, we glued the motors to either side of one end of the ice cream tub, and bolted the caster to the other end. This created a three-wheeled buggy driven by the two motors at the front. We set the Wi-Fi to automatically connect to our network using the WiFi Config tool on the Raspbian desktop.

All motor controllers should come with some test code so that you can make sure everything is working. The software that installs the PicoBorg Reverse drivers will also put an app on the desktop. If you haven't already, you should run that now. Now is also a good time to make sure that both motors are wired the correct way round. With both motors on forward, the buggy should obviously move forward. With motor 1 on and motor 2 off it should turn left, and with motor 2 on and motor 1 off, it should turn right. If this is different on your buggy, you just need to switch the wires around until it works correctly.

Fire up Python

The PicoBorg Reverse software includes a Python module to control the motors, but it doesn't install it to the global Python directory, so it's not available to scripts that are run from other locations. In order to make this module available, you'll have to copy it across yourself with the following code (you may need to adjust the path depending on where you unzipped the install files):

```
sudo cp /home/pi/picoborgrev/PicoBorgRev.py /usr/lib/cd
pymodules/python2.7/
```

We'll use a simple web server to control the buggy. Web servers work by waiting for requests, and then serving web pages based on the request they get. Normally, the request is given in the URL that the website visitor's browser sends to the web server. For instance, if you visit [## Alternatives to the Pi](http://www.linuxvoice.com/wp-</p>
</div>
<div data-bbox=)

The Raspberry Pi is particularly well suited to this project because the camera is well supported and there are plenty of motor control add-ons to provide all the functionality you need. However, it's not the only option. It should be possible to do more or less the same thing on a BeagleBone Black, although you'll have to do a little bit more work to get streaming video set up (there's a guide here: <http://shrkey.com/installing-mjpg-streamer-on-beaglebone-black>). Larger boards such as the Odroid or Udo should work as well, though they'll drain the batteries faster, and their extra processing power isn't really useful for this project.

It should be possible to use a microcontroller such as an Arduino to handle the motor control (though it would be better to use Bluetooth than Wi-Fi in this case). Getting streaming video working with a microcontroller would be challenging, though probably not impossible if you are determined enough. However, you could do this separately using a wireless webcam.



A bit of glue will hold the motors in place, but be careful not to get any on moving parts.

`content/uploads/2014/04/turtle.png` you are requesting the file `/wp-content/uploads/2014/04/turtle.png` from the server www.linuxvoice.com. The server will respond to this request by sending an image from the Python drawing tutorial from Linux Voice issue 2.

Requests don't have to be for files though. The web server can deal with requests however it wants. You can also send bits of data in the URL. These arguments in the URL string come after a question mark and are separated by ampersands. For example, in the URL www.google.co.uk/search?q=linuxvoice, the argument `q` is set to the string "linuxvoice".

We're going to use the Python Tornado web server to use these requests to control the motors on the Pi. You'll need to install this on the Pi with:

```
sudo apt-get install python-tornado
```

The code to control the motors using the PicoBorg Reverse is:

```
import PicoBorgRev
import subprocess
import tornado.ioloop
import tornado.web

maxspeed = 0.3
PBR = PicoBorgRev.PicoBorgRev()
PBR.Init()
PBR.ResetEpo()

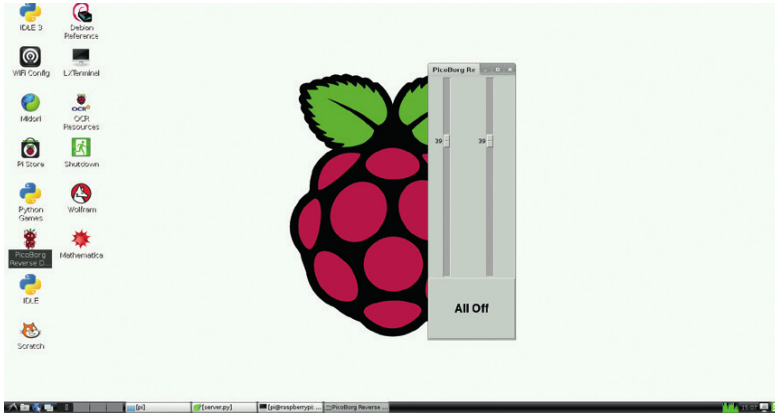
class TurnHandler(tornado.web.RequestHandler):
    def get(self):
        PBR.SetMotor1(min(float(self.get_argument("motor1"))/100, maxspeed))
        PBR.SetMotor2(min(float(self.get_argument("motor2"))/100, maxspeed))
        self.write("Updated")

class HaltHandler(tornado.web.RequestHandler):
    def get(self):
        subprocess.call(["sudo", "halt"])

if __name__ == "__main__":
```

LV PRO TIP

Robots are like Lego: once it's built, play with it for while, then take it to bits and build a new one.



The PiBorg Reverse GUI controller is useful for making sure everything's connected correctly.

```

application = tornado.web.Application([
    (r"/turn/", TurnHandler),
    (r"/halt/", HaltHandler)])
application.listen(8000)
tornado.ioloop.IOLoop.instance().start()
    
```

The final block of this code (which starts with `if __name__`) sets up the web server running on port 8000 (we'll use port 80 – the usual web server port – a bit later). It uses the class `TurnHandler` to handle requests to `/turn/`, and the class `HaltHandler` to deal with calls to `/halt/`. Both of these classes extend `tornado.web.RequestHandler`, which sets them up with almost everything they need. The only thing this code does is add the get method that is called whenever a HTTP GET request is sent to the appropriate URL.

You can access the arguments passed in the URL using the `self.get_argument()` method. The two calls in `TurnHandler` are to get the arguments called `motor1` and `motor2`. We then use these values (which

we'll set between -100 and 100) to set the speed of the motor (which is between -1 and 1). We've limited the motor speed using

the global variable `maxspeed` to stop the motors burning out.

The code here works for a PicoBorg Reverse, but it should be fairly trivial to adapt it to other motor boards. If your motor controller only supports on and off, you'll have to include an if statement to test the

“You could add an output device to the Pi such as a little LCD screen to display the IP address.”

arguments against a threshold, and if it is, turn the motor on. For example:

if float(self.get_argument("motor1")) > 30.0:

#code to turn motor one on

else:

#code to turn motor one off

`HaltHandler` is used to turn the Pi off, since there's no other way to shut it down cleanly when there's not a screen unless you SSH in, which is a little excessive for a simple robot.

We called the file `server.py`, and you can start it running from the LXTerminal command line with:

python server.py

We'll get it running automatically a bit later on.

You can now control the robot from the Raspberry Pi by opening the web browser and going to <http://localhost:8000/turn/?motor1=20&motor2=20> (be careful not to accidentally drive your robot off your desk when testing this). You can then stop the motors by going to <http://localhost:8000/turn/?motor1=0&motor2=0>.

Control from other machines

You can also access this from other computers on the same network by using the IP address of the Pi. To find out the IP address of the Pi, open LXTerminal and type `ifconfig`. This will output a block of information for each of the network interfaces. The one you need is labelled `wlan0`, and you're looking for the `inet addr`. In the following, the IP address is 192.168.0.33:

```

wlan0  Link encap:Ethernet  HWaddr bc:ee:7b:87:7b:38
       inet addr:192.168.0.33  Bcast:192.168.0.255
       Mask:255.255.255.0
       inet6 addr: fe80::beee:7bff:fe87:7b38/64 Scope:Link
       UP BROADCAST RUNNING MULTICAST  MTU:1500
       Metric:1
       RX packets:88425 errors:0 dropped:0 overruns:0
       frame:0
       TX packets:81516 errors:0 dropped:0 overruns:0
       carrier:0
       collisions:0 txqueuelen:1000
       RX bytes:76786575 (76.7 MB) TX bytes:14405224
       (14.4 MB)
    
```

Unfortunately, this isn't fixed and may change from time to time if you reboot the Pi, and it won't be easy to run `ifconfig` if the Pi is mounted inside a robot. There are a few ways around this. Many Wi-Fi routers enable you to assign a static IP address to a device,

Web sockets

The method we've used for controlling the motors is, well, a little hacky. It works, but it doesn't work well. The main problem is that there's a large overhead each time you change the motor speed. The phone app has to negotiate a new TCP connection and send the data, then the Tornado server re-initialises the module to send data to the server. This means there's a noticeable lag between turning the controls and the buggy responding. Part of this is also due to the interval that the app checks the accelerometer, but this has been adjusted to work well with the speed of the server.

A better method would be to create a communications channel through which you can continuously send data. There are a couple of options for this: TCP sockets or Web sockets. Both are supported by Python, and both have plugins for the Cordova framework that we're using for the Android app. Neither should be excessively complex to set up, though they will require some knowledge of both Python and JavaScript. Using one of these methods, you should be able to reduce the latency of the control and increase the frequency with that the app updates the accelerometer readings.

which will enable you to set it so the same IP address will always be assigned to the Pi. You could add some output device such as a little LCD screen to the Pi to display the IP address. The simplest method is to use another Linux computer to scan the address range and find the IP address for the Pi. You can do this using Nmap.

First you'll need to install Nmap from your distro's repositories (on Debian-based systems, this is done with **sudo apt-get install nmap**). Since the above server runs on port 8000, we can use this to detect the Pi. The following command will check all computers in the IP range 192.168.0.0 to 192.168.0.20 to see if that port is open.

```
nmap -sT 192.168.0.0-20 -p 8000
```

The Pi will respond with something like this:

```
Nmap scan report for 192.168.0.33
```

```
Host is up (0.039s latency).
```

```
PORT      STATE SERVICE
```

```
8000/tcp  open  http-alt
```

Usually, the Pi will be the only IP address that returns a state of **OPEN** for this port.

Currently, you also need to start server.py manually. We'll set it to start automatically at the end once everything else is set up.

Getting visuals

Installing the Raspberry Pi camera module is simply a case of slotting it into the correct port (the one between the Ethernet and HDMI ports) with the silver coloured bare metal facing towards the HDMI port, then enabling it. Enter **sudo raspi-config** in LXTerminal, then select Enable Camera, then Yes. You'll need to reboot the Pi for the changes to take effect. There's a video guide at www.raspberrypi.org/help/camera-module-setup if you have any problems.

If you don't have a camera mount to attach to the chassis, a blob of Blu-tack also works.

That's the hardware completely set up. There's still a little bit of software to set up on the Pi, but it doesn't involve any more coding. As the saying goes, "good programmers borrow, great programmers steal", and that's exactly what we're going to do. Streaming video from a Raspberry Pi to a website isn't new, and there's no reason to do it yourself.

The easiest setup we've found is at https://github.com/silvanmelchior/RPi_Cam_Web_Interface. Just download the ZIP file and install it with:

```
unzip Rpi_Cam_Web_Interface-master.zip
```

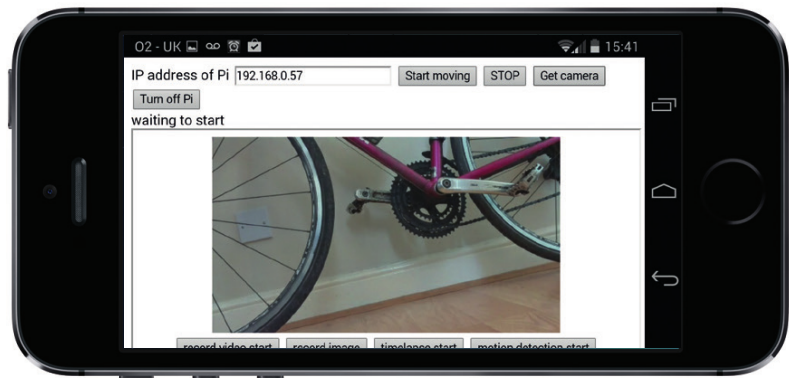
```
cd Rpi_Cam_Web_Interface-master
```

```
chmod a+x RPi_Cam_Web_Interface_Installer.sh
```

```
./RPi_Cam_Web_Interface_Installer.sh install
```

Reboot the Pi so it picks up all the new settings. It'll automatically create a web server (on port 80) that starts when you turn on the Pi, and hosts a website with the streaming video as well as some settings so you can control the video stream (and record pictures and video from your buggy).

Once it's up and running, you should be able to open <http://<ip-address-of-pi>> in a web browser on



The finished app controlling the buggy. It's not much to look at, but the controls are intuitive and fun.

another computer connected to the same network and see the video stream. You don't need to modify it at all, but it'll fit into the smart phone app we'll create in the next step a bit better if you get rid of the title and resize the image.

To do this, open up the **/var/www/index.html** file on the Pi using a text editor running as sudo. For example, to do this in Leafpad, run

```
sudo leafpad /var/www/index.html
```

To get rid of the title, delete the line:

```
<h1>RPi Cam Control</h1>
```

The size you want the image to be will depend on the resolution of your phone screen. We went with a width of 400 pixels, though you can adjust this at the end to make it fit properly on your phone. To do this, change the line:

```
<div><img id="mjpeg_dest"></div>
```

to:

```
<div><img id="mjpeg_dest" width=400px height=auto></div>
```

The only thing left to do set the Python script that runs the motor control server to start automatically (we didn't do this earlier because the setup for the webcam overwrites the file it's done in). Just add the following line (you may have to modify it depending on where you saved **server.py**):

```
python /home/pi/picoborgrev/server.py
```

to the file **/etc/rc.local** directly before the final line (exit 0). Again, you'll need to use a text editor running as superuser, so open Leafpad with sudo as you did with **index.html**. That's all the setup for the Pi – now to create the phone app that will control it.

Hands on

The easiest way to create smartphone apps is with Apache Cordova (as seen in Linux Voice issue 2). The idea is that it enables you to use web technologies (mainly HTML and JavaScript) to create apps that can access phone functions that regular web pages cannot. In this case, we'll access the accelerometer.

Accelerometers measure what's known as proper acceleration. This is a little different from what most people know of as acceleration, because it's the acceleration experienced by an object. This means that an accelerometer resting on a surface will experience an acceleration of 9.8 m/s because it's experiencing that acceleration from gravity. On the

other hand, if you drop the accelerometer, it will read 0 because it's in free fall and not experiencing any acceleration. (Actually, it will read a little higher than 0 because of air resistance.)

As long as you hold the device still, the accelerometer measures gravity. It measures it in three dimensions (x, y and z), which means that you can use it to measure the orientation of the device in three dimensions. In other words, it tells you which way up the device is.

“Cordova’s Accelerometer plugin should work on just about every smartphone.”

First, though, you'll need to set up a Cordova environment on your development machine. According to the Cordova documentation, the

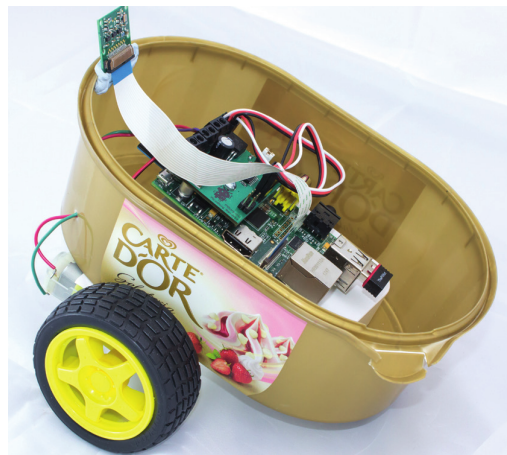
Accelerometer plugin should work on every phone that supports Cordova, which is just about every smartphone (Amazon Fire OS, Android, BlackBerry 10, FirefoxOS, iOS, Ubuntu Touch, Windows phone 7 & 8, Windows 8 and Tizen). We'll look at Android here, and there are details of how to get started in the different environments on the Cordova website (http://cordova.apache.org/docs/en/3.4.0/guide_platforms_index.md.html#Platform%20Guides).

Cordova runs on node.js, so you'll need to install **npm** (the node package manager) from your distro's repositories. People using Ubuntu-based systems will need to add a PPA to get the most up-to-date version of node for this.

```
sudo add-apt-repository -y ppa:chris-lea/node.js
sudo apt-get update
sudo apt-get install npm openjdk-6-jdk
sudo npm install -g cordova
```

As well as Cordova, you'll also need the Android Software Development Kit (SDK) from Google (download this from <http://developer.android.com/sdk/index.html>). Once you've downloaded and installed this, you'll need to set up some environmental variables so that Cordova knows where to find

```
export PATH=${PATH}:/home/ben/adt-bundle-linux-x86-20140321/sdk/platform-tools:/home/ben/adt-bundle-linux-x86-20140321/sdk/tools
```



That's all it takes to build a simple robot: Linux on the Raspberry Pi to power the motors, and Linux on a smart phone to handle the controls.

```
export PATH=${PATH}:/home/ben/adt-bundle-linux-x86-20140321/sdk/platform-tools:/home/ben/adt-bundle-linux-x86-20140321/sdk/tools
```

You'll need to amend the paths to point to the Android SDK you downloaded and extracted. You can run these commands in the terminal, but it won't remember the settings, so you'll have to re-enter them each time you reboot. In order to add these permanently, add the two lines to the **.bashrc** file in your home directory.

To create a Cordova project for the buggy run:

```
cordova create buggy
cd buggy
cordova platform add android
cordova plugin add org.apache.cordova.device-motion
```

We based our code on the **watchAcceleration** Full Example from http://cordova.apache.org/docs/en/3.3.0/cordova_accelerometer_accelerometer.md.html#Accelerometer. This provides everything to read the acceleration periodically, and the function **onSuccess()** is called when it's successfully read.

Before getting into what we do with the acceleration, let's look at how we'll lay out the screen. This is the code between **<body>** and **</body>**:

```
IP address of Pi <input type="text" name="ip" id="ippi">
<button onclick="startMoving()">Start moving</button>
<button onclick="stopMoving()">STOP</button>
<button onclick="getCamera()">Get camera</button>
<div id="sendingstring">waiting to start</div>
<iframe id="cam" width=100% height=600px></iframe>
<iframe id="turniframe" width=1px height=1px></iframe>
```

As you can see, there will be a text field to enter the IP address of the Raspberry Pi, and three buttons to start controlling the motors, stop controlling the motors, and start the camera feed. **<div id="sendingString"></div>** will hold the URL that's being sent to control the motors. This isn't necessary, but it's useful to see what's going on.

Embed video

Iframes enable you to embed web pages inside of web pages. The first one (with the id **'cam'**) holds the streaming video from the Raspberry Pi camera. The second one (with the id **'turniframe'**) doesn't actually hold anything useful, but by changing its URL, we can use it to create GET requests that control the motors.

To make this work, you need three new JavaScript functions that will run when the buttons are pressed:

```
function getCamera() {
    document.getElementById('cam').src = "http://" + document.
    getElementById('ippi').value;
}
function startMoving() {
    window.piMoving=true;
}
function stopMoving() {
    window.piMoving=false;
}
```

The first of these just sets the URL of the **cam** iframe to the address of the streaming webcam

running on the Pi. Remember that we've removed the title and resized the image to make it fit in here. The rest of the controls are still there, so you can tune the streaming image by scrolling down the iframe.

startMoving() and stopMoving()

set the variable **window.piMoving** to **true** or **false**.

This is just a global variable that we'll use to control whether the motor settings are sent to the Pi or not.

You also need to update the **onSuccess()** function (which runs every time it reads the acceleration) to:

```
function onSuccess(acceleration) {
    var element2 = document.getElementById('sendingstring');

    if (window.piMoving) {
        var motor1Prop = (acceleration.y + 10)/20;
        var motor2Prop = 1 - motor1Prop;
        var totalSpeed = acceleration.z * 10;
        var motor1Speed = motor1Prop * totalSpeed;
        var motor2Speed = motor2Prop * totalSpeed;
        sendString = "http://" + document.getElementById('ippi').value + ":8000/turn/?motor1=" + motor1Speed + "&motor2=" + motor2Speed;
        element2.innerHTML = sendString;
        document.getElementById('turniframe').src = sendString;
    }
}
```

Although it's not completely necessary, you can increase the frequency with which the app updates the buggy's speed by altering the frequency setting in the **startWatch** function. In the following example, it updates it once a second, but you could set this to be higher or lower.

```
function startWatch() {
    var options = { frequency: 1000 };
    watchID = navigator.accelerometer.
watchAcceleration(onSuccess, onError, options);
}
```

The full code is on the Linux Voice website.

This calculates the speed for the two motors.

acceleration.y is used to change the direction and **acceleration.z** is used to change the speed. This works for holding the phone in landscape. With the screen at right angles to the ground, the buggy will stop, and as you tilt the screen forward (so the screen starts to face upwards), it will start to move. If you tilt the screen back, the buggy will move backwards. Tilting the screen from side to side (as though it were a car steering wheel) will turn the buggy.

Security

This robot is controlled via Wi-Fi with absolutely no security whatsoever. Anyone else on the network could quite easily take over control. Normally this isn't a problem on a local area network, but there may be occasions where you want a bit more privacy. Tornado does handle security quite well, though it's beyond the scope of this tutorial to go into it in detail. Take a look at the documentation on the project's website for guidance on this (www.tornadoweb.org/en/stable). Securing the video stream may be a little trickier, as it's not really designed for it.

```
index.html (~/.buggy/www) - gedit
index.html (~/.buggy/www) - gedit
index.html
36 window.piMoving=false;
37
38 }
39
40 function haltPi() {
41     document.getElementById('turnIframe').src = "http://" + document.getElementById('ippi').value + ":8000/halt/";
42 }
43
44 function stopWatch() {
45     if (watchID) {
46         navigator.accelerometer.clearWatch(watchID);
47         watchID = null;
48     }
49 }
50
51 function getCamera() {
52     document.getElementById('can').src = "http://" + document.getElementById('ippi').value;
53 }
54
55 function onSuccess(acceleration) {
56     var element2 = document.getElementById('sendingstring');
57
58     if (window.piMoving) {
59
60         var motor1Prop = (acceleration.y + 10)/20;
61         var motor2Prop = 1 - motor1Prop;
62         var totalSpeed = acceleration.z * 10;
```

The acceleration in each axis is returned as a number between -10 and 10. The formula $(\text{acceleration.y}+10)/20$ returns a number between 0 and 1 depending on how far the phone is rotated. This is then used as a multiplier for the speed of one motor. The multiplier for the speed of the other motor is this value taken away from 1.

The overall speed is the acceleration in the z axis multiplied by 10. This gives it the range -100 to +100 (with negative values being backwards). This is the same range that the motors have. To get the final speed for each motor, we just multiply that motor's proportion by the total speed. This is quite a simplistic method of calculating the speed, and the turn directions will go back to front if the phone's held the wrong way up. However, it works, and it's easy to understand, so it's good enough for our buggy.

With the code ready, you just need to get it on to a phone in order to run it. Unfortunately, this can require a little fiddling with the Udev rules. There's full information on the Android developer site here: <http://developer.android.com/tools/device.html>. You can skip step 1 because Cordova will handle it for you.

Once this is set up, and the phone is plugged into your computer, you can compile and transfer it to the phone. Enter the following in a terminal in the root directory of the app:

```
cordova build android
```

```
cordova run android
```

As you can see, this isn't a particularly elegant solution. Running two web servers is a little over the top. It could have been re-written to do everything in one either by serving the video up from Python or by controlling the motors from PHP. The phone app could be more integrated rather than just serving up an iframe of the webcam controller. However, this project isn't about technical perfection, it's about demonstrating how you can quickly and easily link things together to easily create complex robots by using the tools that are available on Linux. 🐧

Ben Everard is the co-author of the best-selling book on learning Python with the Raspberry Pi, *Learning Python with Raspberry Pi*. He wrestles lions for fun.