Digital Signal Processing 2

Power Systems Analysis

Rory Bateman, David Collins

DT021 Year 4

November 6 2013

Contents

List of Figures

List of Tables

1 Introduction

1.1 Criteria for Circuit Analysis

This project entails the use signal processing techniques to analyise the current, voltage and sampling frequency of three different electrical appliances, namely a charger, a monitor and computer rack. Using Matlab we were required develop a function which would automatically detect what appliance was in use when given only the current and voltage waveforms of each appliance, the waveforms were captured at a frequency of 8kHz. It was also required that our function be able to detect an unknown appliance in conjunction with the three existing appliances. The power analysis system we devised must also be able to detect if two appliances are running at the same time and determine which two they are. We were not provided with data for the case of two appliances running simultaneously and had to synthesise our own data using electrical circuit theory. A number of assumptions were taken in the approach we took to solve this task. First we assumed the appliances were supplied by mains voltage which would mean the the voltage was 230V A.C supplied at 50Hz. We also assummed the appliances to be in parallel when more than there was more than one in operation. It is also important to note that the voltage and current waveforms given to us for this project was training data only. Hence the function we devised is able to deal with new data signals that vary slightly from the training data. Chapter two of the report deals with the theory and rational of the approach we took to obtaining a functioning power system analysis tool. Chapter three outlines how we implemented this theory using MATLAB. In chapter four we examine the testing of our system and in particular the design for testing the script. Chapter five takes a look at the results obtained from the project. Finally the report is concluded in chapter six with a summary of the project undertaken and also includeds application scope and performance issues.

1.2 Background

1.2.1 Energy efficiency

In todays economic climate every business and indeed most individuals worldwide are looking at measures that can be taken to reduce costs. One particular area in which people are looking to save costs is their use of electricity. Reducing costs and increasing efficiency has become a major factor when talking about power consumption. Digital signal processing is a very useful tool that can be used to analyse power systems and hence help us achieve our cost saving goals.

The system we have designed allows us to analyse our electricity consumption behaviour and could be used as tool in reducing costs, reducing energy usage and increasing efficiency. Although the function written is only robust to a certain extent the principal can be taken and used in many different applications.

1.2.2 Digital Signal Processing in context

With analog electronics fast becoming dated technology more and more people are turning towards the digital age and hence DSP applications are increasing. Digital signal processors are now widely available and cost effective so it is fair to say DSP will continue to affect engineering design in our modern daily life and can be used for many different applications. This project examines one such application, power system analysis.

Digital Signal Processing in Power System Protection and Control is a book written by Rebizant, Szafran and Wisniewsk in which they bridge the gap between the theory of protection and control and the practical applications of protection equipment. One particular section of the book is of importance and relevant to this project, namely section 3.2.3 Digital Signal Processing. In this section they describe how the protection of a plant or equipment can be implemented using DSP, The protection operation related to determination of the state of the protected plant (faulty or healthy) and issuing the final decision is based on digital processing of sampled input signals, with the aim of respective criteria values[1]. One can see that this project has similarities with the application described above and with development could be used in the same manner for the protection of the appliances in use at any given time.

Another existing technology which has significant relevance to this project of power system analysis using DSP is the area of smart metering. Smart meters are the next generation of electricity meters that allow for remote meter reading, real time pricing and remote operation. The ESBi state that they are simply intelligent two-way communications devices with digital real time power measurement. The principal of our project and the smart meters are similar in the way they digitally process a current and voltage signal and record measurements of the signals.

2 Theory

2.1 AC circuit theory

A simplified circuit diagram is shown in Figure 1. Note the currents indicated are functions of time. If only the rack is connected, Kirchoff's Current Law informs us that the total current will simply be equal to the rack current:

$$i_{total}(t) = i_1(t)$$

, or

$$i_{total}(t) - i_1(t) = 0$$

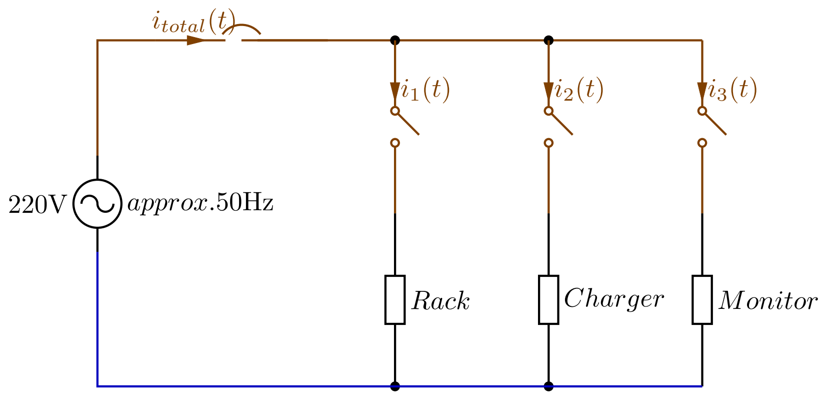


Figure 1: Circuit diagram

. Similarly, if both the rack and the charger are connected, then we expect that

$$i_{total}(t) - i_1(t) - i_2(t) = 0$$

. This elementary observation seems to suggest an easy way to test for appliances: take the current signal you have been provided, subtract the current signals belonging to known appliances and see if the results is equal to zero!

Unfortunately, however, it's not quite that simple. The difficulty arises because we are considering the signals in the time domain and unless the signals are perfectly 'aligned' in time, they won't cancel!

NB: Our method is based essentially on Kirchoff's Current Law.

2.2 Time-domain vs. Frequency-domain representations

So far we have considered the signals in the time-domain. Due to the linearity of the Fourier Transform, the sum of the Fourier transform of two signals is equal to the Fourier transform of the sum. Thus, if

$$i_{total}(t) = i_1(t) - i_2(t) \tag{1}$$

, then

$$I_{total}(\omega) - I_1(\omega) - I_2(\omega) \approx 0$$

. This allows us to test for the presence of certain appliances using the same method we had considered in the time-domain. For example, if the rack and charger are connected, we expect that

$$I_{total}(\omega) - I_1(\omega) - I_2(\omega) \approx 0.$$

.

2.3 Statistical measures

There will always be some variation in our run-time data. The current drawn by our appliances tomorrow, for example, will not be precisely the same as it is today. Thus we need to resort to statistical measures when trying to 'match' our signal data. Three such measures which will be useful for our purposes are the *mean*,

$$\bar{x} = \frac{1}{N}\sum_{i=0}^{N} x[n],$$

variance,

$$\mathrm{Var}(x) = \frac{1}{N-1}\sum_{i=0}^{N}(x[n]-\bar{x})^2,$$

and *standard deviation* — which is just the square root of the variance:

$$\sigma = \sqrt{\mathrm{Var}(x)} = \left\{\frac{1}{N-1}\sum_{i=0}^{N}(x[n]-\bar{x})^2\right\}^{\frac{1}{2}}$$

Qualitatively, the mean corresponds to the intuitive notion of 'average', while variance is a measure of the 'spread' in a sequence of values.

These measures can perhaps best be understood by looking at the probability distribution of the sequence (Figure 4).

2.4 Other methods

There are a multiple ways in which this problemt can solved. In an effort to achieve our goal of obtaing a functioning power system analysis tool we looked at a number of different methods. One such method was the use of corrolation and autocorrelation, another was to obtain the impeadance and use that data to solve the task. Another option considered was to obtain the real power in each appliance and use circuit theory. The final method taken into consideration was to examine the power spectrum of each signal. All of these methods and reasons for going against them are outlined in the appendix (Section B).

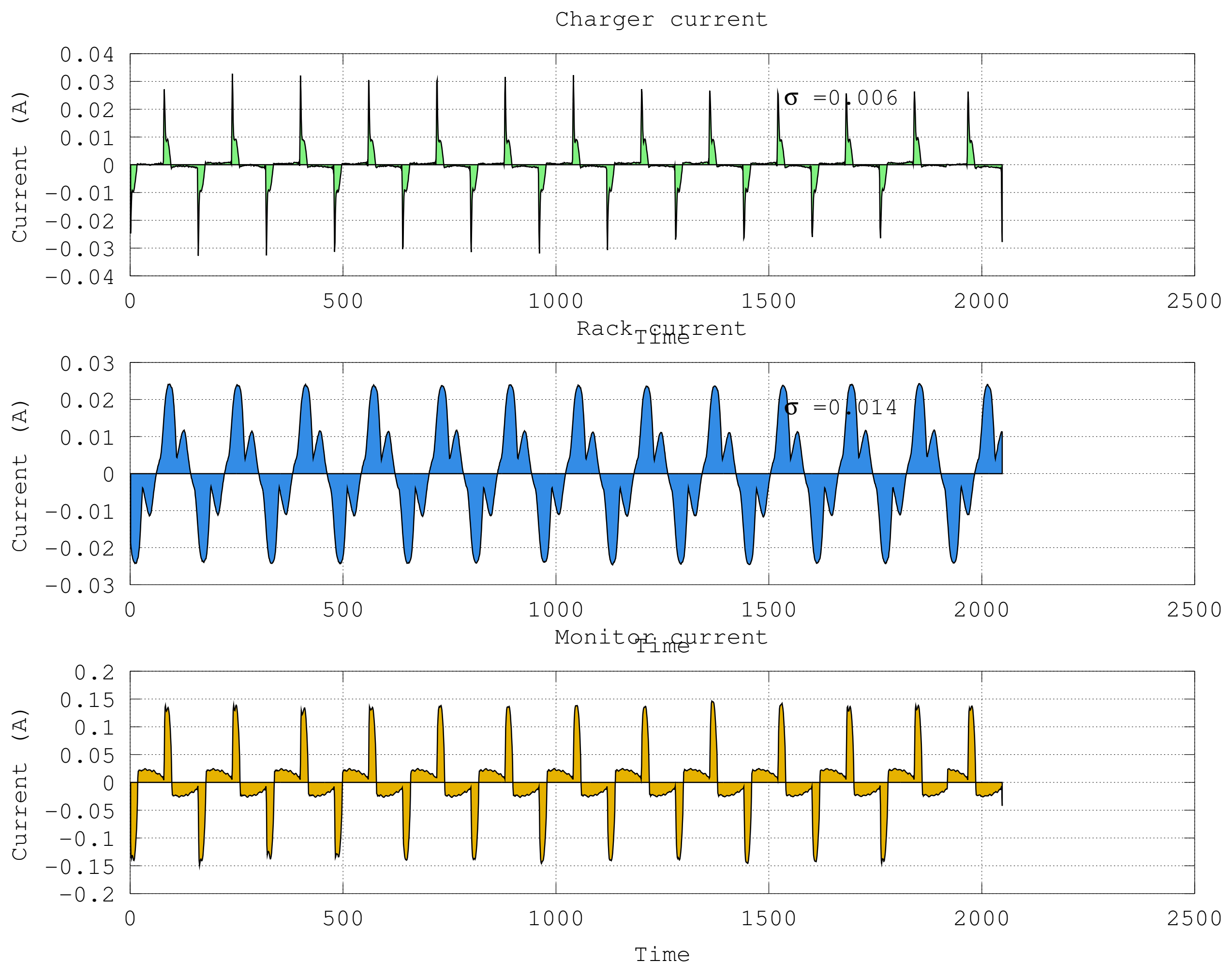


Figure 2: A glance at the currents in the time-domain

3 Implementation

There were a number of potential pitfalls when trying to implement a solution to this problem. These include:

- variations in sampling rate,
- variations in signal length, and
- variations in signal phase.

To allow for different sampling rates, we used MATLAB's resample() function.

```
if (sampling_rate ~= original_sampling_rate)
        current = resample(current,
            original_sampling_rate, sampling_rate
            );
end
```

3.1 Time-shifting

We employed a MATLAB function to compensate for the phase shift in signals. The method we employed to do this can be summarised as follows:

1. Find the maximum value in a single period of the voltage;
2. Iterate sample-by-sample over a single cycle of the voltage to find the first sample close to this magnitude; and
3. circularly shift *both the voltage and current* by this number of samples.

Thus, the main body of the helper function is as follows:

```
for n=range
        if (signal_struct.voltage(n) > (maxVal-
            tol))
                nShift = n-sample_offset;
                break;
```

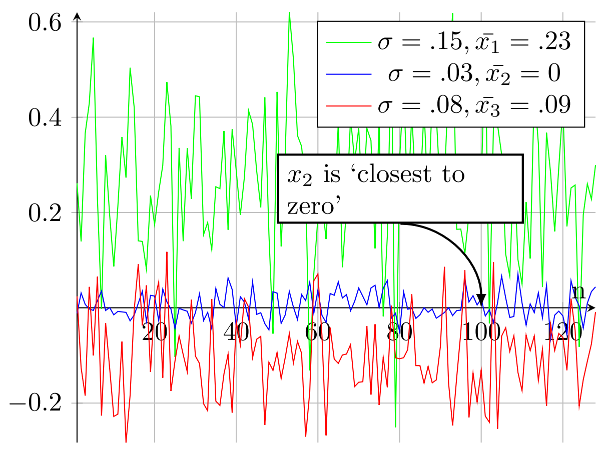


Figure 3: When similar signals are subtracted from each other the result is 'close' (but not equal) to zero.

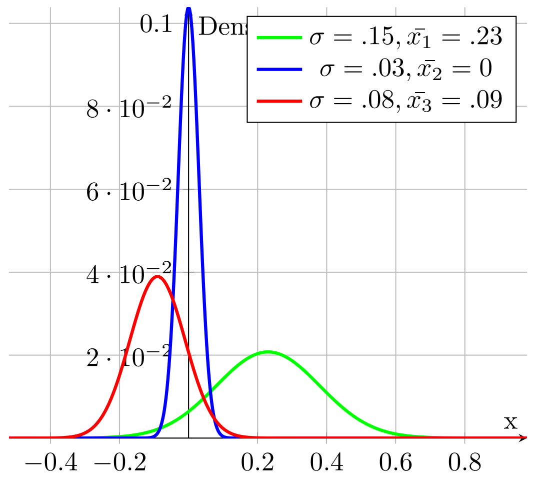


Figure 4: Probability distribution and statistical measures

```
        end
end
```

The full code, which provides a clearer indication of variable meanings, can be found in Listing 3.

This is called in our main application in the manner shown.

```
% 2. Compensate for phase offsets.
appliance = correct_phase_shift(appliance,
    samples_in_a_period);
```

3.2 Variation in run-time data

The current and voltage data retrieved during the running of the application will never match data of the known appliances exactly. Even if the phases of the voltages are aligned correctly, there will always be some sort of variation.

Consequently, when we test for the presence of appliances using an equation such as Equation 1, we can never expect the result to be exactly zero — even when the appliance connected is the one we are testing for. What we need to test for, really, is the degree of 'closeness' to zero. More specifically, we expect two things when we subtract two signals which are almost exactly equal:

1. the average should be very close to zero; and
2. the deviation of the result from its average (i.e. zero) should be almost negligible.

Two statistical measures that test for these characteristics are the 'mean' and the 'standard deviation'. They are employed in our application as follows.

Firstly, we subtract our known currents from the current data in our run-time application — storing the result in a matrix. (Using a matrix simply allows us to perform multiple statistical tests on multiple signals. The same thing could be accomplished using row or column vectors — it would just be more awkward.)

```
% 1. Compare with monitor
currentDiff(:,1) = appliance.current - monitor.
    current;
% 2. Compare with rack
currentDiff(:,2) = appliance.current - rack.
    current;
```

Then we extract various statistics.

```
currentDiffStd = std(currentDiff)
currentDiffVar = var(currentDiff)
currentDiffMean = mean(currentDiff)
```

3.3 Application output

In order for our application to be useful to working engineers, or even just domestic users, the output of our application needs to be reasonably 'user-friendly'. Although our main appliance_detect() function returns only the number corresponding to an appliance (or combination of appliances), our MATLAB test-script maps this number to a more meaningful message. This is done using a simple **switch**() statement which tests for different cases (Listing 1).

The message can then be printed to the user's screen using a command such as

```
msg = strcat('Connected appliance(s): ',
    applianceDescription);
disp(msg);
```

The code of our actual 'appliance_detect' function is shown in Listing 4. As can be seen, the function takes three arguments.

Listing 1: Generating user-friendly application output

```
% Run the actual appliance detection function.
appliance_no = appliance_detect(current, voltage, new_sampling_rate);

% Then map numbers to appliances.
switch(appliance_no)
        case {1}
                applianceDescription = 'monitor';
        case {2}
                applianceDescription = 'rack';
        case {3}
                applianceDescription = 'charger';
        case {4}
                applianceDescription = 'monitor and rack';
        case {5}
                applianceDescription = 'monitor and charger';
        case {6}
                applianceDescription = 'rack and charger';
end
```

3.4 Data structures

For passing arguments between functions, we chose to use data structure in some cases. This structure had two elements — one for the current signal, and one for the voltage signal.

```
appliance.current = current;
appliance.voltage = voltage;
```

This allowed us to perform operations — e.g. phase-shifts — on appliance data by passing just a single argument (rather than two) to the relevant function. Our `correct_phase_shift ()` function, for example, takes the afore-mentioned data structure type as its first argument.

4 Testing

Writing an appropriate test script for our application requires as much care as the design of the application itself, in one sense. We wrote a dedicated MATLAB script for testing our application. It has a number of important features:

1. **noise simulation** the ability to add noise to our synthesized signal;
2. **phase offsets** time-domain shifting of signals by an arbitrary number of samples; and
3. **sampling-rate conversion** in case we want to simulate different sampling frequencies.

Boolean flags in our test script enabled the turning on an off of these features (see Listing 5).

4.1 Noise simulation

Expecting our application to cope with arbitrarily large noise signals and adding them indiscrimately is unrealistic. While ensuring that our application can tolerate some noise in the run-time data, we want to keep our 'signal-to-noise ratio' reasonably high. We did this as follows:

- take the smaller of the amplitudes of the two currents you are using as test data (remember we are *synthesizing* current signals for our tests) — in the case of only current signal just take whatever its amplitude is;
- take some fraction of this — let this be the amplitude of our noise signal;
- use the **rand**() or **randn**() MATLAB function to generate a random signal — ensuring it is the same length as our current test signal; and
- simply add this random signal to our current signal.

```
noiseScalingFactor = min(max(current1),max(
    current2));
noiseAmplitude = noiseScalingFactor / 5;
current = current + randn(length(current),1)*
    noiseAmplitude;
```

4.2 Phase offsets

To introduce a phase offset in the synthesized data, we simply circularly-shifted it in the time-domain.

```
if (phaseShift)
        nShift = 23;
        current = shift(current, nShift);
        voltage = shift(voltage, nShift);
end
```

A complementary `correct_phase_shift ()` routine compensates for this in the actual running of the application (cf. Section 3.1).

Listing 2: Sampling-rate conversion

```
if (samplingRateConversion)
        new_sampling_rate = 8e+3;
        % If we want to change the sampling rate (even just for testing purposes)
        % we will need to 'resample'.
        if (sampling_rate != new_sampling_rate)
                disp('Resampling ...');
                current = resample(current, new_sampling_rate, sampling_rate);
                voltage = resample(voltage, new_sampling_rate, sampling_rate);
        end
        sampling_rate = new_sampling_rate;
end
```

4.3 Re-sampling

To make allowances for different sampling rates, we used MATLAB's built-in resample() function (Listing 2).

4.4 Speed

We are also interested in how long the detection algorithm requires to return its results. To measure this we used MATLAB's built-in tic() and toc() functions. We insert them in our test script as follows:

```
tic();
appliance_no = appliance_detect();
elapsed_time = toc();
```

This information might be useful for debugging purposes (or even for the end-user), so let's print it to the screen.

```
printf('(Appliance detection algorithm took %.5f
    seconds\n)', elapsed_time);
```

5 Results

5.1 Pass/fail tests

The results of pass-fail tests for various scenarois — simulating the connection of different appliances and different signal-to-noise-ratios — are summarised in Table 1. As is clear from the table, the results are satisfactory. The algorithm detects the correct appliance for effectively all realistic scenarios — excluding only those in which the signal-to-noise ratio is unreasonably low.

5.2 Speed

On average, the actual detection algorithm (i.e. ignoring the signal synthesis performed in the test-run scripts — which wouldn't be required in run-time) required approx. 28 milliseconds to return its results. The results of several test-runs are shown in Table 2. This information is important to consider, as a real-world application might have to compute results for large amounts of data 'on-the-fly'.

5.3 Output captures

The results of simulating the rack and charger being connected are shown in Figure 5 — 7.

6 Summary

Using only simple circuit theory (e.g. Kirchoff's Current Law) and some basic statistical measures, we have managed to devise a potentially useful signal processing application.

6.1 Application scope

The appliances considered for this application are relatively simple. Even though they are non-linear, their current draw and power dissipation are largely predictable. Many modern appliances could prove significantly more challenging to identify. The current draw of appliances such as DVD players, desktop PCs, and laptops, for example, is highly unpredictable. Although not necessarily impossible to accommodate in this sort of application, their identification would almost certainly require methods beyond those outlined in this report.

6.2 Performance issues

As noted in Section 4.4, our detection algorithm delivered its results with reasonable speed. This was in a 'laboratory' scenario, however, and we need to bear in mind that a real-world application might require faster speeds.

6.2.1 Real-time performance

An important consideration for an application such as this is its ability (or inability) to perform in 'real-time'. If the software is expected to detect certain appliances 'on the fly' rather than using historical data more attention will need to be given to the efficiency of the algorithms used.

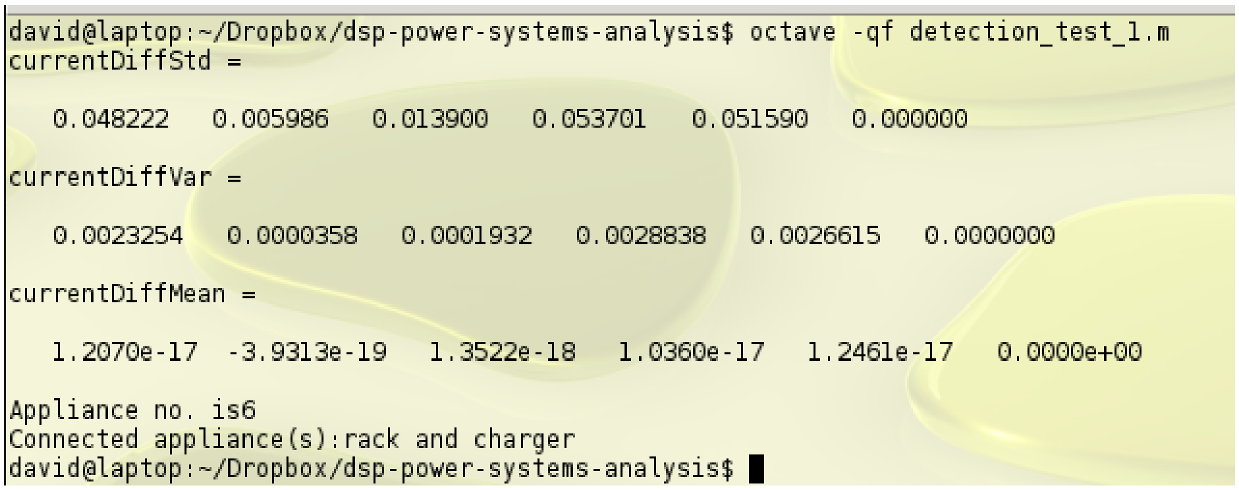
Signal-to-noise ratio	5	4	3	2	1	.75	.5	.4	.3	.2
Monitor	🟩	🟩	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Rack	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
Charger	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟥
Monitor + rack	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
Monitor + charger	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟥
Rack + charger	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟥

🟩 = pass, 🟥 = fail

Table 1: Pass/fail tests for different scenarios

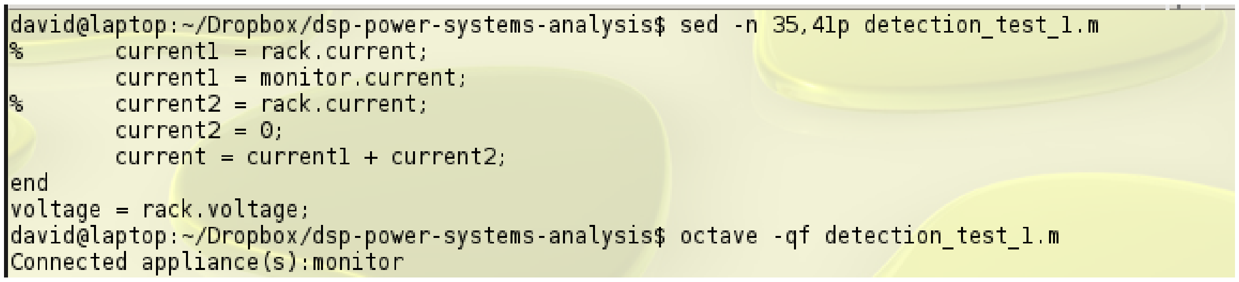
Run	1	2	3	4	5	6	7	8
Duration (ms)	27.90	27.89	27.69	27.82	27.99	27.78	27.71	28.03

Table 2: Speed tests



```
david@laptop:~/Dropbox/dsp-power-systems-analysis$ octave -qf detection_test_1.m
currentDiffStd =

   0.048222   0.005986   0.013900   0.053701   0.051590   0.000000

currentDiffVar =

   0.0023254   0.0000358   0.0001932   0.0028838   0.0026615   0.0000000

currentDiffMean =

   1.2070e-17  -3.9313e-19   1.3522e-18   1.0360e-17   1.2461e-17   0.0000e+00

Appliance no. is6
Connected appliance(s):rack and charger
david@laptop:~/Dropbox/dsp-power-systems-analysis$ 
```

Figure 5: Results of simulating rack and charger being connected

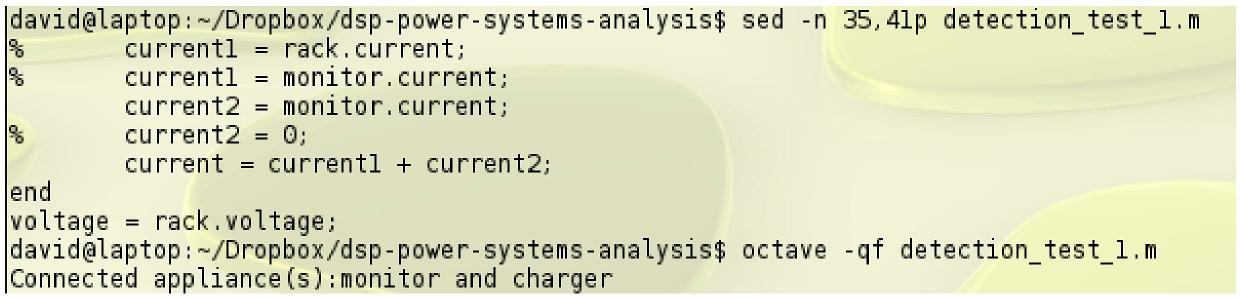


```
david@laptop:~/Dropbox/dsp-power-systems-analysis$ sed -n 35,41p detection_test_1.m
%       current1 = rack.current;
        current1 = monitor.current;
%       current2 = rack.current;
        current2 = 0;
        current = current1 + current2;
end
voltage = rack.voltage;
david@laptop:~/Dropbox/dsp-power-systems-analysis$ octave -qf detection_test_1.m
Connected appliance(s):monitor
```

Figure 6: Results of simulating monitor being connected

6.2.2 MATLAB vs. other programming languages

MATLAB is a useful development tool, and enables a programmer to test ideas easily and quickly. As it is an interpreted rather than compiled language, however, it has a serious drawback: programs written in MATLAB are slow. For optimum performance, a developer will probably choose to write their program in C or C++. This has the added advantage that users without MATLAB can run the program also.



```
david@laptop:~/Dropbox/dsp-power-systems-analysis$ sed -n 35,41p detection_test_1.m
%       current1 = rack.current;
%       current1 = monitor.current;
        current2 = monitor.current;
%       current2 = 0;
        current = current1 + current2;
end
voltage = rack.voltage;
david@laptop:~/Dropbox/dsp-power-systems-analysis$ octave -qf detection_test_1.m
Connected appliance(s):monitor and charger
```

Figure 7: Results of simulating charger and monitor being connected

A MATLAB code

A.1 Phase-shift compensation algorithm

As mentioned previously, it is crucial that the phases of the appliance voltages are aligned in order for our method to correctly. We tried two approaches to tackling this problem.

- **Phase-shift compensation in the frequency domain** This approach involved taking the Fourier Transform of the voltage signal, finding the frequency bin corresponding to 50 Hz, and taking the angle of the corresponding value. Ideally, this angle determines how much we need to 'rotate' or 'shift' our signals. Unfortunately, this method proved unreliable in practice — working only approx. two out of every three times. This may be due to the fact that the voltage signals were not perfect sinusoids.

- **Phase-shift compensation in the time domain** In this method we simply found the sample corresponding to the maximum amplitude of the voltage signal within one cycle, and shifted the signal by this amount.

The code we used in the final implementation is shown in Listing 3.

Listing 3: Compensating for phase shift in MATLAB

```
% Circular shift waveforms to remove phase shift (with relation to a cosine waveform).

% This is done as follows:
%               1. Find the maximum value of the signal. This maximum can be found within a certain period;
     then, as long as the signal is periodic (with no exponential decay or increase), the local maximum
    should be approximately equal to the global maximum.
%               2. Iterate over the samples in a single period, stopping as soon as you find one close to
    the maximum value.
%               3. The sample number at which you stopped is the number of samples you need to shift.
function new_signal_struct = correct_phase_shift(signal_struct, samples_in_a_period)
        N = length(signal_struct.voltage);
        % Take a snapshot of the signal slightly offset from its beginning.
        % This is to allow for slight perturbations in the first few samples.
        % Make sure that the offset is an integer multiple of the number of samples
        % in a period.
        sample_offset = samples_in_a_period*2;
        range = sample_offset:sample_offset+samples_in_a_period;
        % Find the max, min, and amplitude.
        maxVal = max(signal_struct.voltage(range));
        minVal = min(signal_struct.voltage(range));
        amplitude = maxVal - minVal;
        % The tolerance we refer to when searching for a sample 'close' to the
        % maximum depends on two factors:
        %               1. the number of samples in a period, and
        %               2. amplitude of the waveform.
        angle_per_sample = 2*pi / samples_in_a_period;
        tol = amplitude * (1-cos(angle_per_sample));
        nShift = 0;
        for n=range
                if (signal_struct.voltage(n) > (maxVal-tol))
                        nShift = n-sample_offset;
                        break;
                end
        end
        signal_struct.voltage = shift(signal_struct.voltage,-nShift);
        signal_struct.current = shift(signal_struct.current,-nShift);

        new_signal_struct = signal_struct;
end
```

A.2 Detection algorithm

The code for our actual detection algorithm is presented in Listing 4.

Listing 4: Main 'appliance_detect' function

```
% Authors: Rory Bateman & David Collins
% Date: October 28 2013
% Description: This function detects the presence of a particular appliance on
% a mains AC circuit based on the three parameters provided (current, voltage, and sampling-rate).
% Return value: The returned value is an integer, which has the following meaning:
%        0 - unknown
%        1 - monitor
%        2 - rack
%        3 - charger
%        4 - monitor + rack
%        5 - monitor + charger
%        6 - rack + charger

function appliance_no = appliance_detect(current, voltage, sampling_rate)
        % Load known current signals (time domain and frequency domain)
        load('currentData.mat');
        matchTol = .07;

        appliance.current = current;
        appliance.voltage = voltage;

        original_sampling_rate = 8e+3;
        fundamental_freq = 50;
        original_signal_length = 2048;

        samples_in_a_period = original_sampling_rate / fundamental_freq;

        appliance_no = 0; % Assume an unknown appliance until proven otherwise

        % 1. Check if signals have the same sampling rate. If not, resample.
        if (sampling_rate != original_sampling_rate)
                current = resample(current, original_sampling_rate, sampling_rate);
        end

        % 2. Compensate for phase offsets.
        appliance = correct_phase_shift(appliance, samples_in_a_period);

        % Compare signals in frequency domain
        % Note that we are adding two current waveforms in some cases to simulate
        % the connection of two appliances. This is legitimate as long as we have
        % compensated for any phase offsets between the voltages that generated these currents.

        % We will place the results in a matrix.
        % This is purely for convenience. It makes the extraction of
        % statistical measures - such as variance, standard deviation, and
        % mean - somewhat easier.
        N = length(appliance.current);
        nTests = 6;
        currentDiff = zeros(N,nTests);

        % 1. Compare with monitor
        currentDiff(:,1) = appliance.current - monitor.current;
        % 2. Compare with rack
        currentDiff(:,2) = appliance.current - rack.current;
        % 3. Compare with charger
        currentDiff(:,3) = appliance.current - charger.current;
        % 4. Compare with monitor and rack combined
        currentDiff(:,4) = appliance.current - (monitor.current + rack.current);
        % 5. Compare with monitor and charger combined
        currentDiff(:,5) = appliance.current - (monitor.current + charger.current);
        % 6. Compare with rack and charger combined
        currentDiff(:,6) = appliance.current - (rack.current + charger.current);

        currentDiffStd = std(currentDiff);
        currentDiffVar = var(currentDiff);
        currentDiffMean = mean(currentDiff);
        [minVal, appliance_no] = min(currentDiffStd);

        save('statisticalData.mat', 'currentDiff');

        % If there was very little correlation with _any_ signal
        % assume an unknown appliance (=> appliance_no = 0).
        if (minVal >= matchTol)
                appliance_no = 0;

end
```

A.3 Test script

Listing 5: MATLAB test script

```
% Title: Appliance detection test-runs
% Author(s): David Collins and Rory Bateman
% Date: October 31 2013
% Description: Test runs for the 'appliance_detect' function.

% Load known current data
load('currentData.mat');

% RUN-TIME PARAMETERS
%
% Boolean flags to control certain run-time parameters
%
% Enable noise simulation?
noiseSimulation = 1;
plotNoise = 0;
signalToNoiseRatios = [5,4,3,2,1,.75,.5,.4,.3,.2];
%signalToNoiseRatio = 5;
% Enable sampling-rate conversion?
samplingRateConversion = 0;
% Enable a phase-shift?
phaseShift = 1;
% Simulate an unknown signal?
unknownSignal = 0;

%for index = 1:length(signalToNoiseRatios)
signalToNoiseRatio = signalToNoiseRatios(3);
% TEST SIGNAL SYNTHESIS
%
% Synthesize test signals
% current = randn(length(rack.current),1);
if (unknownSignal)
        current = randn(length(rack.current),1);
else
        current1 = charger.current;
%       current1 = rack.current;
%       current1 = monitor.current;
        current2 = monitor.current;
%       current2 = 0;
        current = current1 + current2;
end
voltage = rack.voltage;

% NOISE HANDLING
%
% Test to see how much noise the appliance can cope with in the signal.
%
% Expecting it to cope with a noise signal whose amplitude is greater than
% the amplitude of the signals themselves is ambitious. Hence, we limit
% the amplitude of our noise signal according to the size of the currents themselves.
%
% Note: There is no point in adding noise to the signal if it is already unknown!
if (noiseSimulation && !unknownSignal)
        if (var(current2) == 0)
                noiseScalingFactor = max(current1);
        else
                noiseScalingFactor = min(max(current1),max(current2));
        end
        noiseAmplitude = noiseScalingFactor / signalToNoiseRatio;
        orig_current = current;
        current = orig_current + randn(length(current),1)*noiseAmplitude;
        if (plotNoise)
                % Compare original and noisy signals
                subplot(3,1,1);
                plot(orig_current);
                subplot(3,1,2);
                plot(current, 'r');
                subplot(3,1,3);
                plot(current, 'b');
                hold on;
                plot(orig_current, 'g');
                pause();
        end
end

% PHASE COMPENSATION
%
% Ensure that the application can cope with
% phase offsets (i.e. shifts in the time-domain).
```

```
if (phaseShift)
        nShift = 23;
        current = shift(current, nShift);
        voltage = shift(voltage, nShift);
end

% SAMPLING RATE COMPENSATION
%
% What is the sampling rate of our data?
sampling_rate = 8e+3;
if (samplingRateConversion)
        new_sampling_rate = 8e+3;
        % If we want to change the sampling rate (even just for testing purposes)
        % we will need to 'resample'.
        if (sampling_rate != new_sampling_rate)
                disp('Resampling ...');
                current = resample(current, new_sampling_rate, sampling_rate);
                voltage = resample(voltage, new_sampling_rate, sampling_rate);
        end
        sampling_rate = new_sampling_rate;
end

% IMPORTANT PART
% Run the actual appliance detection function.
%
% Note: We also time how long the algorithm takes - using tic() and toc().
tic();
appliance_no = appliance_detect(current, voltage, sampling_rate);
elapsed_time = toc();

% Then map numbers to appliances.
switch(appliance_no)
        case {0}
                applianceDescription = 'unknown';
        case {1}
                applianceDescription = 'monitor';
        case {2}
                applianceDescription = 'rack';
        case {3}
                applianceDescription = 'charger';
        case {4}
                applianceDescription = 'monitor and rack';
        case {5}
                applianceDescription = 'monitor and charger';
        case {6}
                applianceDescription = 'rack and charger';
end

% OUTPUT
%
% Print results on screen.
%msg = strcat('Appliance no. is ', num2str(appliance_no));
%disp(msg);

msg = strcat('Connected appliance(s): ', applianceDescription);
disp(msg);

%printf('(Appliance detection algorithm took %.5f seconds\n)', elapsed_time);

%end
```

B Other Methods

B.1 Correlation

We also considered using correlation to identify signals. All of the known currents have strong sinusoidal components, however, which poses a problem. Cross-correlating a sinusoidal signal with another sinusoidal signal with the same frequency components yields a periodic sequence.

B.2 Impedance

Another method we considered was identifying appliances by their impedance. After all, for a linear electrical circuit, the impedance is simply a transfer function the ratio of the complex voltage to the complex current.

Unfortunately, the circuits in question are not linear there are clearly frequency components in the current which are not present in the voltage input. This is evident from a quick glance at Figure 8, in which the frequency content of the current waveforms for each appliance are plotted above the corresponding voltage frequencies.

Consider the electrical components typically present in a charger, for example. The input stage of the charger will typically include a transformer and a rectifier. The rectifier consists of diodes, which exhibit non-linear behaviour.

B.3 Power Spectrum Density

The third technique we considered using to solve this task was to find the power spectrum density of each signal from the appliance and compare them. Power Spectral Density is the counterpart in the frequency domain of the autocorrelation function (ACF), which is a function of the time shift variable m. The power spectrum can be used to indicate how the signals power is distributed in the frequency domain. . Our goal was to extract information such as the signals power or energy distribution in the frequency domain which can be obtained using the power spectrum method which in turn would allow us to differentiate between appliances in use at any given time.

However when trying to implement this method we found it to be complex and time consuming. One issue that arose was the ability to differentiate between the different power spectrums of each appliance as the characteristics of the each signal are quite similar.

B.4 Power used by appliances

A fourth alternative method was taken into consideration for this problem which was to obtain the power in watts being used by each appliance. As we know from basic circuit theory the real power being used can be defined as $P = VI\cos(\theta)$. Or for a purely resistive load we have:

$$P = VI = I^2R.$$

However problems arose when we looked into using this method such as calculating the phase angle between the voltage and current and also the appliances not being purely resistive loads.

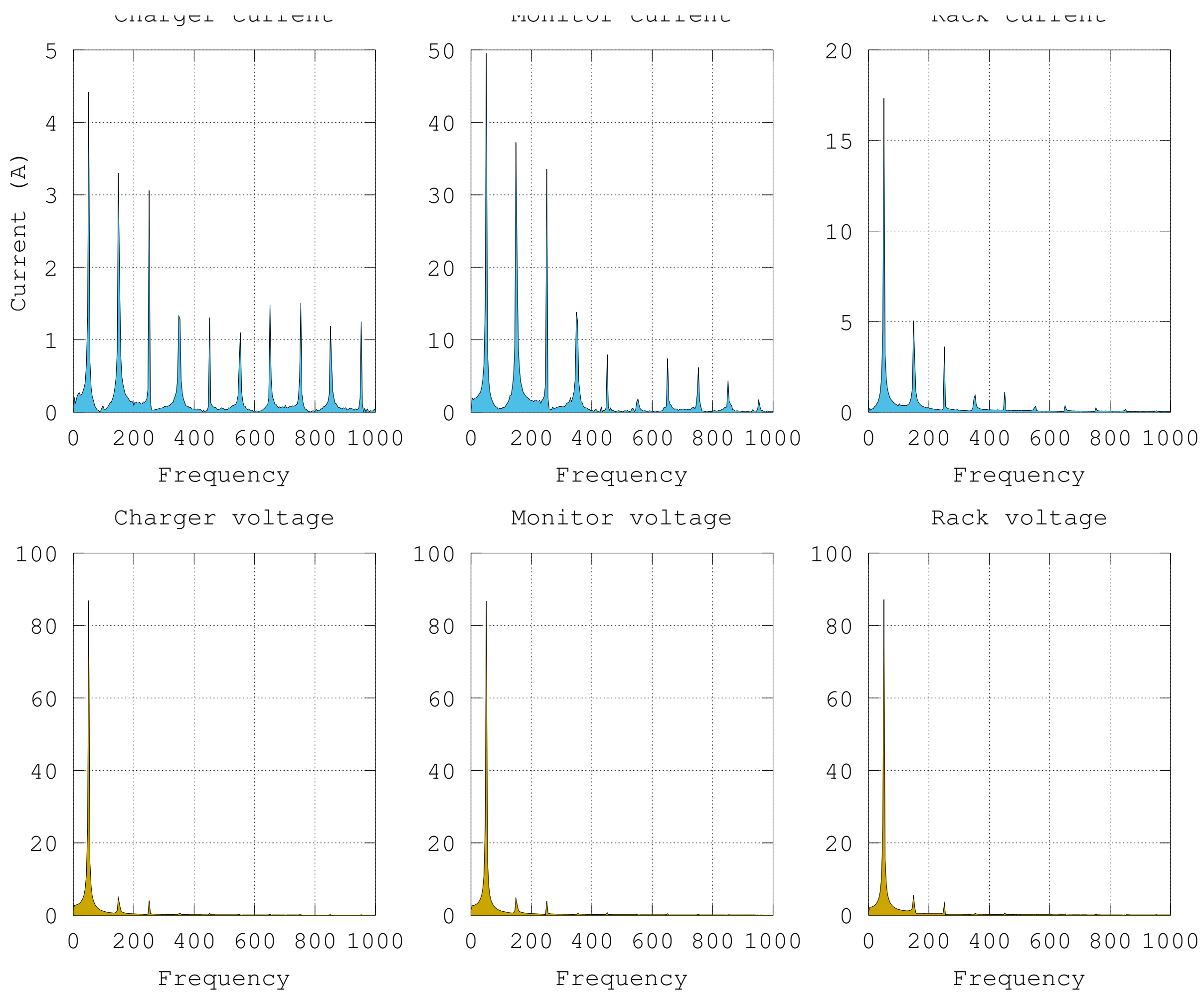


Figure 8: Voltages and currents in the frequency domain