Imperial College London
Department of Computing

Disaggregating Smart Meter Readings using Device Signatures

by

Daniel A. Kelly (aka Jack)

Submitted in partial fulfilment of the requirements for the
MSc Degree in Computing Science of Imperial College London

September 2011

Abstract

Wise management of electricity consumption is becoming increasingly important. Domestic electricity prices in the UK increased by 35 % over the past 8 years; an upward trend which is unlikely to reverse any time soon. As well as the economic costs, the environmental problems associated with the combustion of fossil fuels are becoming increasingly detrimental to the well-being of many species, including our own.

One of the cheapest and easiest ways to improve energy efficiency is to monitor one's electricity consumption using a type of smart meter called a "home energy monitor" available for around £40. These display whole-house consumption in real-time, allowing the user to learn which devices and behaviours consume the most energy. Studies have demonstrated that the feedback provided by energy monitors can enable users to reduce consumption by 5-15 %. If every domestic user in the UK reduced electricity consumption by 10 % then the UK's annual CO_2 output would be reduced by 6 million tonnes. However, the information displayed by home energy monitors is not as useful as it could be.

Home energy monitors measure *aggregate* consumption for an entire building. Research has demonstrated that *disaggregated* information describing appliance-by-appliance energy consumption is more effective than aggregate information. Disaggregation was first successfully implemented in the 1980s using sophisticated smart meters which provide rich, multi-dimensional raw sensor data. However, these sophisticated meters are not readily available at present.

The main contribution of this project is to design, implement and evaluate three different algorithms for disaggregating the sparse, one-dimensional data output by inexpensive, popular and readily available home energy monitors. Each design is trained to recognise a device on the basis of one or more raw recordings of single-device power consumption (which we call *signatures*). The system is then given an unseen aggregate signal from which it must infer the start times, run duration and energy consumption of each device activation present in the aggregate data.

The first design is a simple prototype which uses a *least mean squares* approach to matching a signature to an aggregate signal. As expected, this design succeeds only in a very limited set of contexts but the development of this prototype provided valuable insights. The second design represents a novel approach to determining a set of device *power states* by identifying peaks in the histogram of the device's signature but this design proved to not be ideal.

The final design presented in this dissertation represents a novel approach to disaggregation whereby each device is modelled as a *power state graph*. Each graph vertex (node) represents a power state and each edge represents the conditions necessary to transition from the source to the destination power state. During disaggregation, every possible "fit" of the power state graph to the aggregate data is represented as a tree structure where each tree vertex represents a power state and each tree edge represents the likelihood for the corresponding power state transition. For each complete "fit" of the power state graph to the aggregate data, the disaggregation algorithm reports the start time, the average likelihood, the estimated run time and the estimated total energy consumption. This design has several attractive features: 1) it handles both simple devices and complex devices whose signatures contain power state sequences which repeat an arbitrary number of times; 2) it is probabilistic; 3) a power state graph can be learnt from one or more signatures and 4) the design estimates the energy consumed by each device activation.

This design was evaluated by training it to recognise four real devices: a toaster, a kettle, a washing machine and a tumble drier. The system was then given 13 days of real aggregate data. For all devices except the tumble drier it detected every device activation in the aggregate data although there were a few false positives. The system failed to model the tumble drier, possibly because the drier has very rapid state transitions.

Acknowledgements

I would like to thank my parents for letting my wife and I live at their house while our own home was uninhabitable due to building work. I did a lot of the coding for this project whilst sat in my parents' dining room and, on sunny days, in the back garden. Both my parents provided many stimulating ideas for my MSc project. All the raw data for this project were collected at my parents' house.

I would also like to thank my MSc project supervisor, Dr Will Knottenbelt for his constant support, infectious enthusiasm and lightning-fast responses to my emails. Dr Knottenbelt's excellent "Intro to C++" course at the start of the MSc was a genuine joy to attend, was very instructive and instilled in me a great affection for C++ (although I know I still have a great deal to learn!) After programming in Java and Matlab during the 2nd term of the MSc, returning to C++ for my project felt like "coming home".

I must thank my golden retriever, Scout. Taking her for walks every day provided a healthy break from coding. It's also fair to say that I did 90 % of the design whilst out on walks. I've lost track of the number of times I became stuck on a problem which felt unsolvable whilst coding but a solution revealed itself somewhere in Peckham Rye; usually between the duck pond and the playing fields.

Finally, I must thank my wonderful wife, Ginnie. She has provided constant support, love and affection. Despite being heavily pregnant, she took on the vast majority of the work load associated with managing the building work on our home, allowing me to focus on my project. She's an amazing woman and I'm very lucky to have found her.

Dedication

This project is dedicated to my daughter, Olive, born on the 27th August 2011 (two weeks before this report's deadline!)

Contents

1. Introduction 7
1.1. Background 8
1.1.1. The importance of reducing energy demand 8
1.1.2. Energy consumption behaviour 11
1.1.3. Two types of smart meter 12
1.1.4. Existing disaggregation techniques 13
1.2. Broad aims and research direction 17
1.3. Dissertation outline 17
1.4. Terminology used in this report 18

2. Setting up the measurement and logging equipment 19
2.1. Recording whole-house aggregate power consumption 19
2.2. Recording device signatures 20

3. Early prototype 21
3.1. Prototype version 1 21
3.2. Prototype v2: compensating for dropped measurements 22
3.3. Prototype v3: offsetting & compensating for sample-rate mismatch 22
3.3.1. Offsetting 22
3.3.2. Sample-rate mismatch 23
3.3.3. Accuracy of prototype v3 23
3.4. Further experiments: finally achieving automated alignment of all devices 23
3.4.1. Kettle and toaster 23
3.4.2. Washing machine 23
3.5. Failure to generalise 24
3.6. Prototype summary: lessons and limitations 25

4. First design: histograms and power states 27
4.1. Aims 27
4.2. Broad design principals 27
4.3. Which programming language? 27
4.4. Training strategy 27
4.5. Disaggregation strategy 29
4.6. Implementation 29
4.6.1. Doxygen HTML code documentation 29
4.6.2. Extracting a set of *power states* from the raw signature 29
4.6.3. Extracting a *power state sequence* 33
4.6.4. Data output 33
4.7. Flaws in the "histogram" design 34

5. `gnuplot` template instantiation system 35
5.1. Implementation 36
5.1.1. `gnuplot` output 36
5.2. Limitations and future work 37

6. Final design iteration: graphs and spikes 39
6.1. Introduction 39

6.2. Design . . . 41
 6.2.1. Overview of training algorithm . . . 41
 6.2.2. Overview of disaggregation algorithm . . . 44
6.3. Implementation . . . 49
 6.3.1. Maintaining "legacy" functions and classes . . . 49
 6.3.2. Boost graph library . . . 50
 6.3.3. Overview of the `PowerStateGraph` class . . . 52
 6.3.4. Updating statistics . . . 54
 6.3.5. Non-zero standard deviation & calculating likelihood . . . 54
 6.3.6. Indexing aggregate data by timecode . . . 55
 6.3.7. Data output . . . 55
 6.3.8. Boost program options . . . 56
 6.3.9. Refinements . . . 56
 6.3.10. Parameters . . . 57
 6.3.11. Testing and debugging . . . 58
6.4. Performance . . . 60
 6.4.1. `10July.csv` aggregate data . . . 60
 6.4.2. 3 days of aggregate data (`earlyJuly.csv`) . . . 69
 6.4.3. 10 days of aggregate data (`earlyAugust.csv`) . . . 69
 6.4.4. Conclusions . . . 69

7. Conclusions and future work 71
7.1. Limitations . . . 71
7.2. Other applications of this work . . . 71
7.3. Further work . . . 72
 7.3.1. Combine all three approaches . . . 72
 7.3.2. Machine learning approaches . . . 72
 7.3.3. More efficient implementation & parallelisation . . . 73
 7.3.4. Try a simpler approach based on non-invasive load monitoring . . . 73
 7.3.5. Remove deprecated sections of code . . . 73
 7.3.6. Capture relationships between devices . . . 73
 7.3.7. Capture the probability that a device is active at a given time of day . . . 73
 7.3.8. Ultimate aim: to build a smart meter disaggregation web service . . . 74
7.4. Conclusion . . . 75

A. User guide 77
A.1. Input files . . . 77
A.2. Output files . . . 78
A.3. Configuration file . . . 78
A.4. GNUplot templates . . . 78
A.5. Runtime dependencies . . . 78
A.6. Compiling from source . . . 78
 A.6.1. Compilation dependencies . . . 79
A.7. Generating Doxygen documentation . . . 79
A.8. Running unit tests . . . 79

B. Further (simplified) code listings 81
B.1. Training code . . . 81
B.2. Disaggregation code . . . 82

C. Software engineering tools used 85

Bibliography 85

1. Introduction

Consider someone who has just received a painfully expensive electricity bill. She decides to make a concerted effort to do a better job of managing her electricity usage. What information can she use in order to determine the best course of action? She would probably find it useful to know how much energy each appliance consumes. Even more useful would be the knowledge that, for example, if she replaced her twenty year-old fridge with a newer model then it would pay for itself in two years. The aim of this project is to design and implement a software tool which "disaggregates" the whole-house electricity consumption data recorded by widely available metering hardware; hence providing easy access to "appliance-by-appliance" energy consumption data.

Managing electricity consumption as wisely as possible is becoming an increasingly pressing issue. Yet many energy consumers struggle to manage consumption effectively because their intuition fails to provide an accurate estimate of the quantity of energy each appliance uses [41]. Evidence shows that energy feedback information provided by smart meters can enable consumers to reduce consumption by 5-15 % [29] and that "appliance-by-appliance" disaggregated information is more useful than aggregate information [31]. If every domestic user in the UK reduced electricity consumption by 10 % then not only would they reduce their bill by 10 % but six power stations could be closed, reducing the UK's CO_2 output by six million tonnes per year.

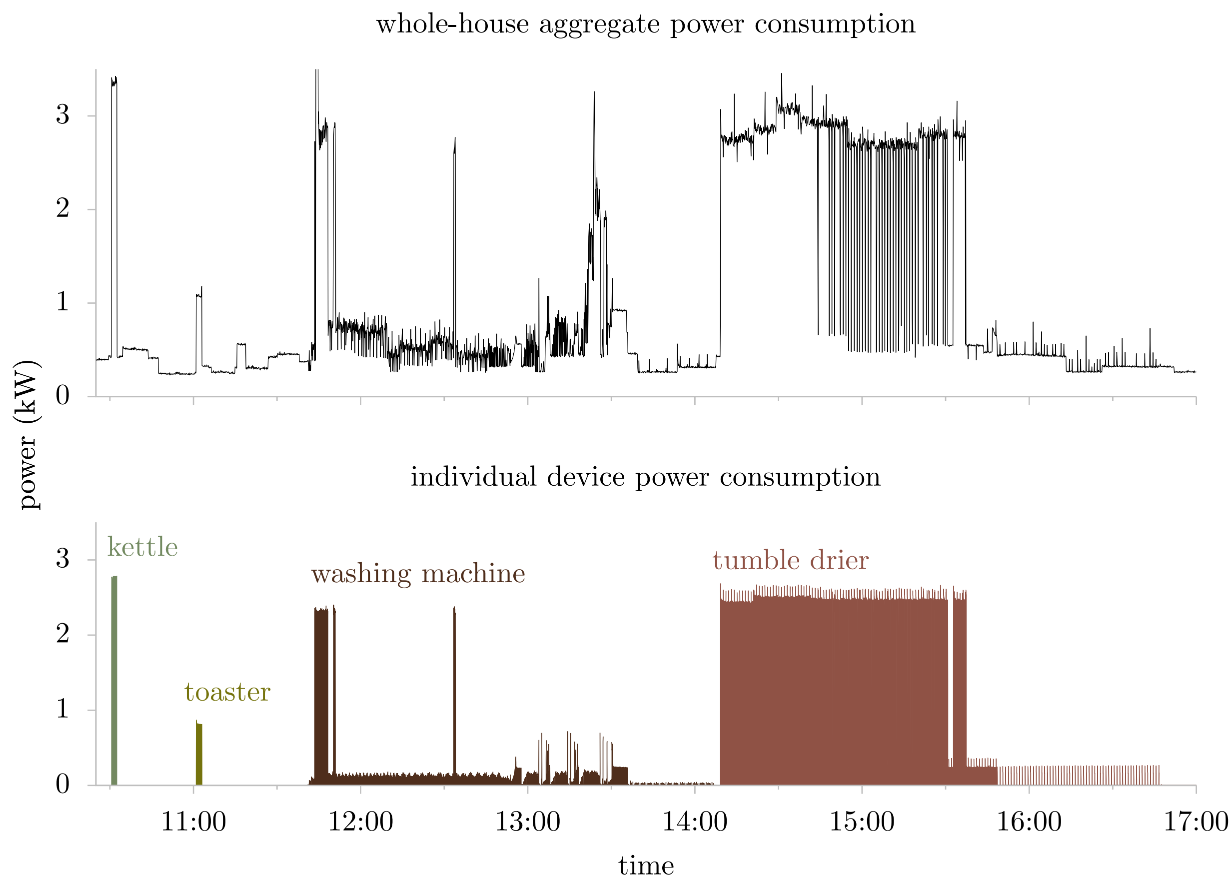


Figure 1.1.: An afternoon's electricity consumption in a domestic dwelling. The top panel shows whole-house aggregate power consumption. The lower panel shows the individual power consumption of four devices. The data in the top and bottom panels were recorded simultaneously using two meters.

Figure 1.1 demonstrates the task at hand. The "whole-house aggregate" plot shows the electricity consumption of an entire domestic dwelling measured using a smart meter installed near the fuse box. The power consumption of individual appliances is plotted in the lower panel.

1.1. Background

1.1.1. The importance of reducing energy demand

It is becoming increasingly important to better manage electricity consumption. Domestic electricity prices in the UK increased by 35 % (in real terms) between 2003 and 2011 [15]. Energy price rises alone are projected to inflate the UK consumer prices index by 1.5 % in Q4 2011 [17], making energy one of the most dominant upward forces acting on UK consumer prices. This upward price trend is likely to continue as global energy demand, especially from non-OECD countries, continues to grow. In 2000, China used half as much energy as the USA. In 2009, China overtook the USA to become the world's largest energy user [11]. The International Energy Agency projects that Chinese consumption will increase by 75 % between 2008 and 2035, and that Indian energy consumption will more than double over the same period (although India will still consume less energy in 2035 than either the USA or China) [11].

The UK is especially exposed to energy price rises. The UK makes extensive use of natural gas for heat and power generation. Between 1995 and 2004 the UK was not just self-sufficient for gas but it was also a net *exporter* of natural gas pumped from the North Sea [20]. Gas production from the North Sea peaked and began its terminal decline in 2000 hence the UK has been a net *importer* since 2004 [20], requiring us to buy increasing amounts of gas on the volatile global market. Gas production from the North Sea halved between 2000 and 2010 (see figure 1.2)[1].

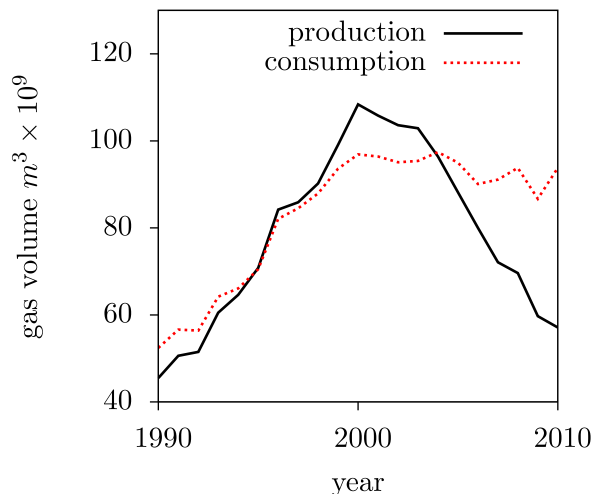


Figure 1.2.: UK natural gas production and consumption. Data from BP Statistical Review [20].

Market fundamentals are not the only upward pressure on energy prices. Many governments now deploy policy mechanisms to encourage individuals and organisations to move away from polluting power sources. Coal, which has historically been the cheapest form of energy, is becoming economically unattractive due to the costs of scrubbing sulphur dioxide from the flue gasses and, soon, capturing and storing CO_2. The UK's Climate Change Act [6], passed into law in November 2008, commits the UK to an 80 % reduction in carbon emissions by 2050 (compared with 1990). Energy taxes in the UK include the Climate Change Levy [13] and the Renewables Obligation [3].

The production of electricity also has environmental costs. In 2010, 75 % of the UK's electricity was produced from hydrocarbon-fired power stations [14], producing 191 million tonnes of CO_2. This represents 39 % of the UK's total CO_2 emissions, making electricity generation the largest single source of CO_2 in the UK [21]. The domestic sector is the single largest load on the electricity grid, accounting for 31 % of the UK's electricity demand [14]. Whilst there has been increasing media attention on CO_2, it is worth briefly considering some peer-reviewed evidence not least because media coverage of the issue is often more noise than signal.

[1] Recent research suggests the UK may have substantial "unconventional" gas reserves in the form of on-shore *shale gas*. The British Geological Survey believes the UK may be sitting on 150×10^9 m^3 of shale gas [19]. However, to put this into perspective, the UK consumes 100×10^9 m^3 of gas per year, so our shale gas reserves will not fundamentally change our energy mix (unlike in the USA where shale gas has been a "veritable game changer": shale gas currently accounts for 23 % of US gas supply and is projected to grow to 40 % by 2030 [22]).

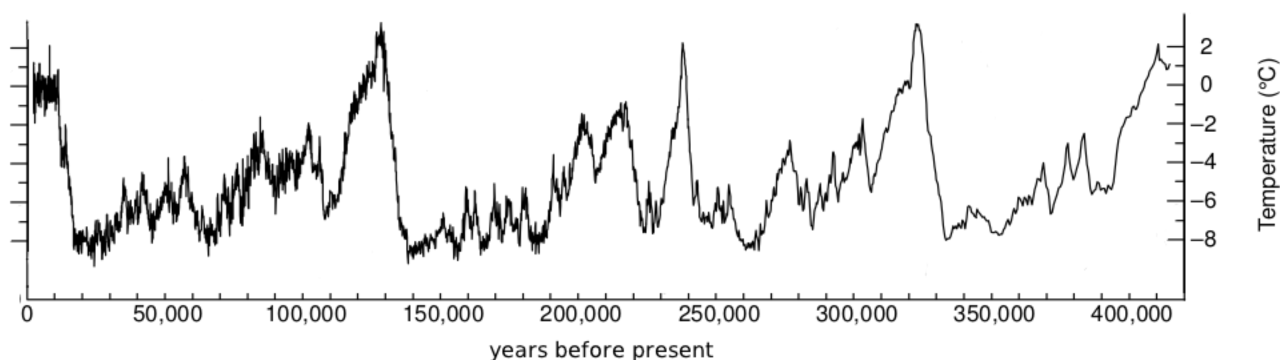


Figure 1.3.: A history of earth's surface temperatures for the past 420,000 years, adapted from [51]. Note the unusually narrow temperature range over the past 10,000 years.

Human activity worldwide currently produces 29 billion tonnes of CO_2 per year [5]. Since the industrial revolution began in 1880, atmospheric CO_2 concentration has increased from 280 parts per million to 391 ppm in 2011 [4], higher than any time during the last 800,000 years [45], and likely higher than any time in the past 20 million years [50] (to put this into some context: anatomically modern humans are hypothesised to have first appeared around 200,000 years ago [46] and agriculture began around 10,000 years ago [35]). This increased CO_2 concentration has two main detrimental effects on the environment: it enhances the "greenhouse effect" [68, 5] by reducing the rate at which the planet loses infrared radiation into space and it acidifies the oceans [25, 48], damaging marine ecosystems.

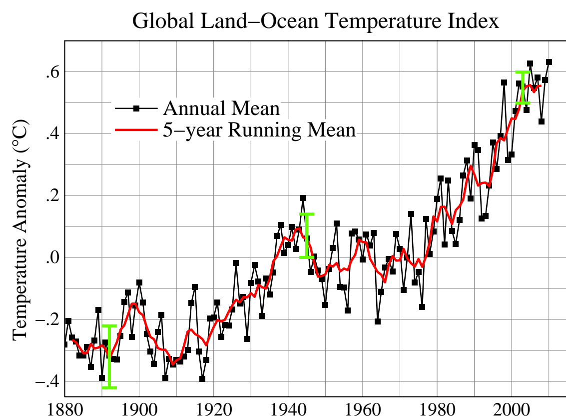


Figure 1.4.: A recent history of global surface temperatures, taken from [16].

There is a considerable body of evidence indicating that the world's systems are already changing. For example: the amount of infrared radiation emitted from the top of the atmosphere into space is decreasing [37] (consistent with the hypothesis that increased atmospheric CO_2 concentration scatters more IR back to earth's surface), global land-ocean temperatures increased by 0.9°C over the period 1880 to 2010 (figure 1.4) [36, 16], ocean heat content is increasing [55], Arctic sea ice is decreasing in extent [40] and volume[2] (see figure 1.5) [56, 12], the Greenland and West Antarctic ice sheets are losing mass [26], the tropical belt is widening [57], plant populations are moving to higher altitudes [44], seasons are starting earlier [62], sea level rose 195mm between 1870 and 2004 and the rate of sea level rise is accelerating [27].

Projections for future climate come with considerable uncertainty but ensemble modelling suggests that, if we continue on current emissions trajectories, the world is likely to warm by at least 2°C by 2100 and possibly by as much as 6°C [54, 5]. The last time the planet was 2°C warmer than today was 10 million years ago. This warming is likely to have many adverse effects, not least of which are the desertification of many currently arable areas [7] and a probable sea level rise of 0.75 to 1.9 meters by 2100 [33, 65, 53]. Palaeoclimatology studies and modelling suggest that many of these effects are likely to worsen over subsequent centuries and will be irreversible for at least the next 3,000 years [61, 54]. (For a recent review of climate science, see [7].)

[2] There is some uncertainty over the thickness (and hence the volume) of the Arctic sea ice. Ice *extent* is relatively easy to measure from satellite imagery but *thickness* is tricky to measure by satellite. The data shown in figure 1.5 are from a *model* of sea ice. Empirical measurements taken by the US Navy [12] and Polarstern icebreaker research vessel [18] suggest the ice may be thinner than the model suggests but these measurements cannot provide complete geographical coverage. Better ice thickness data will be available soon from the CryoSat-2 satellite, launched in April 2010 and currently in a calibration phase.

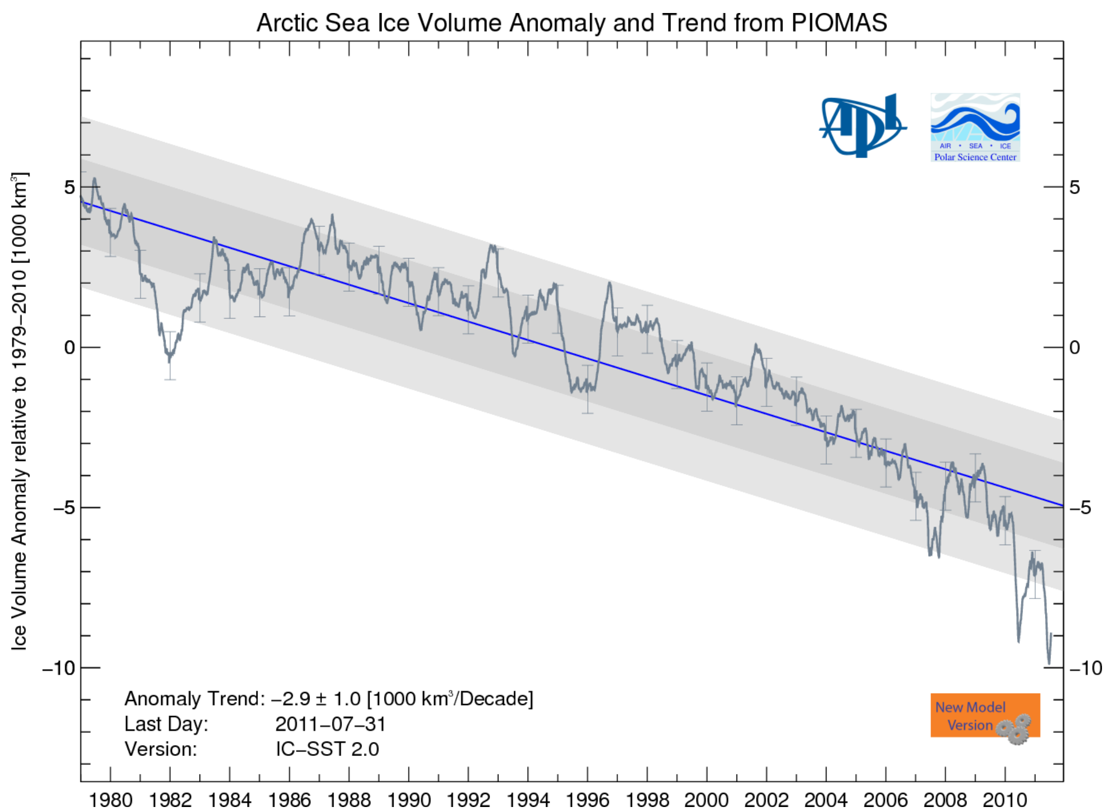


Figure 1.5.: "*Arctic sea ice volume anomaly from PIOMAS updated once a month. Daily Sea Ice volume anomalies for each day are computed relative to the 1979 to 2010 average for that day of the year. The trend for the period 1979- present is shown in blue. Shaded areas show one and two standard deviations from the trend. Error bars indicate the uncertainty of the monthly anomaly plotted once per year.*" Taken from [56, 12]. This ice loss is unexplainable by any of the known natural variabilities [52]. If this trend continues then we will see an ice-free summer Arctic within the next 30 years [66]. Mean total Arctic ice volume was about 15,000 km^3 from 1979-2010; so a y-axis value of about "-15" on the graph above corresponds to zero ice volume.

What effect might this have on humans? Human technological evolution over the past 10,000 years has taken place in a remarkably stable climate (see figure 1.3) [51, 54]. It has been proposed that this stable climate aided human societal development [28]. Unfortunately, paleoclimate data also demonstrates the stability experienced over the past 10,000 years is the exception rather than the norm [28] and that human emissions may push the climate away from this stable state [54] into a state never experienced by the human species, let alone our 21st-century infrastructure.

Is there a "safe" amount of warming? At the 2009 United Nations Climate Change Conference in Copenhagen, 138 countries signed an agreement stating that actions should be taken to keep the temperature increase below 2°C. The 2°C target is a compromise. Even with just 1.5°C of warming, it is extremely likely that low-lying islands will be submerged. But, above about 2.5°C [7] is the temperate range that studies suggest could initiate the total melting of Greenland's ice sheet which contains enough water to raise sea levels by 7 meters [24]. By how much do we need to reduce emissions in order to provide a reasonable chance of keeping warming below 2°C? Figure 1.6 illustrates that, if we want a 75 % chance of limiting warming to 2°C then global emissions need to peak by 2020, decline rapidly and then go negative in 2070! It's an extremely tough challenge but it is technologically feasible.

What can be done to minimise the risk of climate change? Humanity's technological sophistication may provide a key to decouple GDP growth from carbon emissions, allowing economies to grow whilst maintaining a favourable environment. The UK, for example, has already reduced its carbon emissions[3] by 21 % since 1990, a decrease achieved partially through energy efficiency measures (and a shift away from coal-fired power generation).

[3] Although cynics would say that the UK's emissions reduction is due to the UK effectively off-shoring heavy industry.

In summary: there are multiple reasons to manage energy consumption as intelligently as possible. Increased energy efficiency may help to ensure our quality of life remains high despite growing costs of energy. The smart meter disaggregation techniques discussed in this report aim to provide energy consumers with practical information to help them better manage their consumption.

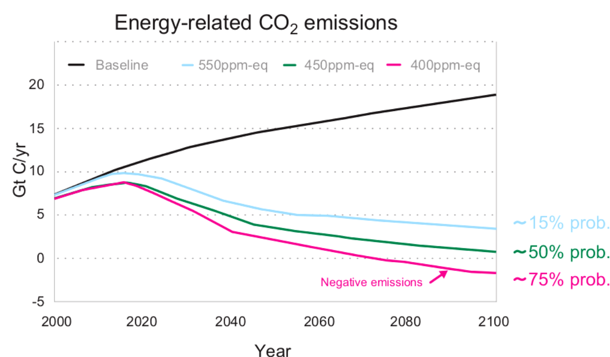


Figure 1.6.: "*Energy-related emission trajectories from 2000 to 2100 to achieve stabilisation of greenhouse gases in the atmosphere at three different targets (coloured lines). The black line is a reference trajectory based on no climate policy. Estimated (median) probabilities of limiting global warming to maximally 2°C are indicated for the three stabilisation targets.*" Taken from [7]

1.1.2. Energy consumption behaviour

Let us assume that people are motivated to improve their energy management. Do they have a sufficiently quantitative understanding of their energy consumption to prioritise correctly?

Prior to the availability of mains energy supplies, most individuals would have had an intuitive, quantitative understanding of the amount of energy consumed by the household. If the stove needed more fuel then someone had to manually shovel solid fuel into it; you couldn't help but notice how much energy was being consumed. In this situation, most individuals would have an intuitive feel for how much energy it took to, say, heat the living room for one evening or cook one meal.

In today's industrialised societies, we do not have such a concrete, tangible understanding of the amount of energy we use. It is a miracle of civil engineering that the energy equivalent of 3 tons of coal[4] is delivered into our homes every year without any noise, any manual labour, any dust, any inconvenience for us.

When we turn on an electrical device, it just works, without any indication of how much energy it is consuming. Hence, when faced with rising electricity bills, we struggle to prioritise correctly when deciding which devices to turn off.

Figure 1.7.: One "unit" of energy, prior to mains utilities.

Studies on residential energy users show that the vast majority are poor at estimating either the consumption of individual devices or total aggregate consumption. Residents often underestimate the energy used by heating and overestimate the consumption of *perceptually salient* devices like lights and televisions [41]. Residents' failure to correctly estimate energy consumption leads to higher total consumption.

How significant is occupant behaviour in determining total energy usage? Energy use can differ by two or three times among identical homes with similar appliances occupied by people from similar demographics [60, 67, 58]. These large differences in energy consumption are attributed to differences in consumption behaviour. If the home provided better feedback about which devices used the most energy then users could tweak their behaviour to make more efficient use of appliances.

Studies have investigated which types of energy feedback information displays are most successful in altering behaviour. Fischer [31] found that "the most successful feedback combines the following features: it is given frequently and over a long time, **provides an appliance-specific breakdown**, is presented in a clear and appealing way, and uses computerized and interactive tools." (my

[4] Average UK household energy consumption: 20500 kWh gas + 3300 kWh electricity = 2.38 MWh total energy ≡ 8.568×10^{10} Joules. Heat content of coal is roughly $3 \times 10^7 \, J/kg$. $\frac{8.568 \times 10^{10} \, Joules}{3 \times 10^7 \, J/kg} = 2865 \, kg$ coal.

Figure 1.8.: A typical "clamp-style" smart meter. The red unit on the left clamps around the incoming mains supply to a building (picture from CurrentCost.com).

emphasis). Darby [29] reports that direct feedback normally reduces energy consumption by 5-15 %. Disaggregated data is also of use to utility companies as it helps with load forecasting.

Providing consumers with disaggregated consumption data can play a part in decreasing primary energy demand. But before we discuss disaggregating an aggregate energy signal, we need to consider how to record an aggregate signal in the first place.

1.1.3. Two types of smart meter

There are two broad types of "smart meter".

The first type is increasingly called a "home energy monitor" rather than a "smart meter". These typically cost around £40 and are bought, installed, maintained and used by consumers. An example is show in figure 1.8. Several utility companies offer these devices for free. These home energy monitors do not need to be installed by an electrician. Instead, they work by "clamping" a sensor around a single mains supply cable at the fuse box. This sensor does not make electrical contact with the mains cable; instead it surrounds the insulated mains cable with a wire coil. Alternating current flows through the mains cable, which produces an alternating magnetic field around the cable. This magnetic field, in turn, induces an electrical current in the home energy monitor's sensor. Hence the sensor is able to measure current passing through the mains cable. This data is typically sent wirelessly to a display unit (up to 20 meters away) which allows the user to see how much power the house is currently using. With a little bit of experimentation, it is possible to get an intuitive feel for roughly how much power each device around the house uses by switching the device on and off whilst watching the display.

The second type of smart meter replaces the "spinning disk" electromechanical utility meter currently installed in UK domestic dwellings. This smart meter is paid for, installed by, and primarily of benefit to the utility company. These "proper" smart meters maintain two-way communication with the utility company. One advantage to utility companies is that they no longer need to send a person to read consumers' electricity meters because the smart meter transmits usage data automatically to the utility company.

Utility-installed smart meters may be a necessary component of a "smart grid" (although alone

do not constitute a smart grid). For example, it is envisioned that smart meters could allow owners of electric vehicles to profit by pushing energy *back into the grid* when demand for electricity (and hence price) is especially high [34]. Why might this be a good idea? The UK plans to make extensive use of wind power to generate electricity. The main problem with wind, of course, is that it cannot be scaled up in response to increased demand. Hence some form of storage is required to capture electrical energy when supply is plentiful, and to assist the grid when demand is greater than supply. Electric vehicles have huge batteries; batteries which, on mass, are large enough to make a meaningful contribution to the grid when demand spikes.

The UK government wants all homes to have utility-installed smart meters by 2019 [9]. British Gas plans to have two million smart meters installed by 2012 [8].

This current project uses the "clamp" style home energy monitor because these are readily available, cheap and around two million units have been installed[5].

Home energy monitors only provide an aggregate energy reading, but we saw in section 1.1.2 that consumers benefit most from disaggregated data. How can we disaggregate the data?

1.1.4. Existing disaggregation techniques

The conceptually most straight forward way to determine how much power individual appliances consume is to install separate meters on each device throughout the house. Such appliance-meters are available from companies like AlertMe for £25 each. Installing separate meters on each appliance is an expensive and inconvenient option for users who want to monitor every appliance in the house. Can we use computational techniques to disaggregate the data recorded by a single smart meter measuring the whole-house consumption?

Conditional Demand Analysis (CDA)

Field surveys have historically been used to acquire details about occupant behaviours. Survey data, along with aggregate energy consumption data, can then be used as the input to *conditional demand analysis* (CDA). The conditional demand analysis technique takes as its input the aggregate data from a sample of homes and uses regression based on the presence of end-use appliances (as reported by occupants) to generate a simple model describing aggregate energy usage as a function of appliance usage. CDA is *time dependent*: it attempts to determine when appliances tend to be used during the day (e.g. that toasters tend to be used in the morning). CDA regresses total dwelling energy consumption onto a list of owned appliances. The determined coefficients represent the usage level and power rating of each device [64].

CDA was developed by Parti and Parti in 1980 [49]. Their regression equations, one for each month of a year of billing data, take the form

$$E_{mo} = \sum_{i} \sum_{app} c_{app,i}(V_i P_{app})$$

where E_{mo} is the monthly electrical energy consumption; P_{app} is a variable indicating the number of appliances; V is a set of interaction variables with elements, i, such as the floor area, number of occupants and income; and c is a regression coefficient.

In order to produce reliable results, the CDA technique requires data from hundreds or even thousands of homes. Even with large datasets, CDA still produces relatively inaccurate results because many end-uses share temporal load profiles and because the surveys tend to be inaccurate [32]. See [64] for further discussion of the CDA technique.

Non-invasive load monitoring

Utility-installed smart meters can measure several variables at once: current, voltage, *reactive power* and *real power*. Let us quickly describe the difference between reactive power and real power

[5] currentcostgroup.com

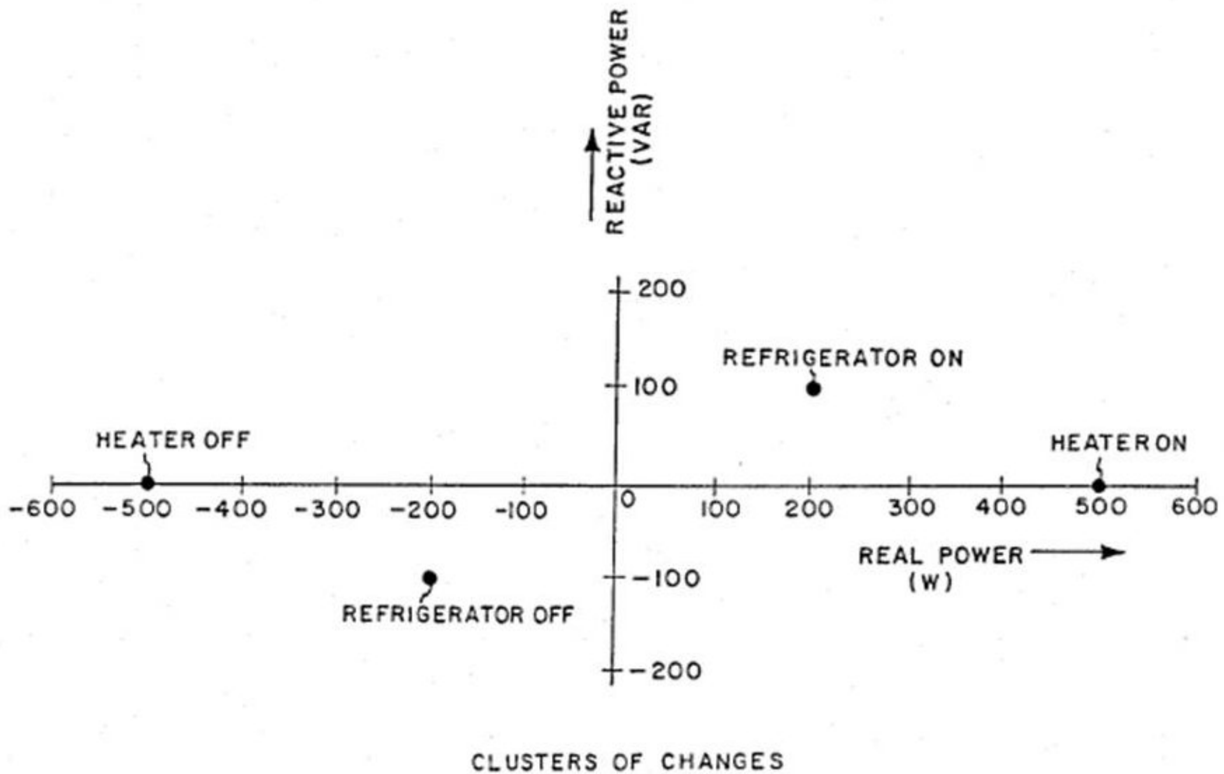


Figure 1.9.: Distinguishing between a heater and a fridge by comparing real and reactive power consumption. The heater is a purely resistive load and hence pulls no reactive power. The refrigerator mostly pulls real power but also pulls some reactive power. These two variables allow us to discriminate between most devices. This diagram is taken from US patent 4858141, filed by George Hart and colleagues from MIT in 1986 [39].

because this difference lies at the heart of the "classic" disaggregation technique called "non-invasive load monitoring".

Mains is an *alternating current* source. If the load is a simple *resistive* load like a heater then current and voltage are perfectly in phase (as you might expect). Resistive loads obediently accept changes in the supply voltage without opposition because they have no ability to store charge.

Some loads oppose the change inherent in an AC supply. These loads transiently store charge because they have a *capacitance* or *inductance.* This stored charge causes them to *react* to change in the supply. If the load is purely *reactive* then voltage and current are 90 degrees out of phase. The product of voltage and current is positive for half a cycle and negative for the other half, hence no net energy flows to the device (but energy is lost in the wiring). No practical devices are *purely* reactive.

Most devices draw both *real power* and *reactive power.* Different devices draw different proportions of real and reactive power. Hence, if we measure both real and reactive power then we can plot devices on a two-dimensional plot like the one in figure 1.9. This powerful tool for distinguishing between different devices lies at the heart of the "non-invasive load monitoring" (NILM) technique, invented by George Hart, Ed Kern and Fred Schweppe of MIT in the early 1980s [38].

The NILM algorithm starts by detecting *change* in the measured parameters. These changes represent transitions between nearly constant steady-state values. For example, consider a 500 watt heater turning on and off. The real power reading will *increase* by 500 watts the moment the heater turns on and will *decrease* by 500 watts the moment the heater turns off.

Changes with equal magnitudes and opposite signs are then paired. In our example, the +500 watt event corresponding to the heater turning on would be paired to the -500 watt event corresponding to the heater turning off and hence the NILM algorithm can deduce the length of time the device has been active and the total energy consumption for each device. The technique is, apparently, sufficiently sensitive to be able to distinguish between different 60 watt bulbs because one bulb

might draw 61 watts while another draws 62 watts.

The original NILM algorithm (which has been successfully implemented commercially) consists of the five steps outlined in algorithm 1.1.

Algorithm 1.1 The original NILM algorithm

1. An "edge detector" identifies changes in steady-state levels
2. Cluster analysis is performed to locate these changes in a two-dimensional *signature space* of real and reactive power (figure 1.9).
3. Clusters of similar magnitude and opposite sign are paired. This catches simple, "two-state" loads like heaters which can only be either *on* or *off* but fails to fully pair clusters attributable to complex devices like dish washers which have many states.
4. Unmatched clusters are associated with existing or new clusters according to a best likelihood algorithm. This step is known as *anomaly resolution*.
5. Events are assigned human-readable labels (e.g. "kettle" rather than "load#213") by matching events to a database of known device power consumption learned during a training phase.

Complex appliances like washing machines are modelled as finite state machines. A washer typically has a heater and a motor which turn on and off in sequence. These are identified as a cluster. The end result is that "washing machine" power consumption can be reported to the user rather than "heater" and "motor" power consumption.

NILM may also make use of a database of existing device signatures in combination with a pattern matching algorithm in order to classify devices.

The original NILM algorithm (algorithm 1.1) makes several assumptions. The first assumption is that measurements of both *real* and *reactive* power are available. The second assumption is that every appliance can be uniquely identified from its position in signature space. This second assumption used to be valid for domestic situations and invalid for commercial situations. However, with increasing numbers of devices in the home, the second assumption is becoming less valid for the domestic situation too.

One challenge for the NILM algorithm is that several classes of device all include heaters (e.g. kettles, washing machines, dish washers, tumble driers etc). These heaters tend to be hard or impossible to distinguish in signature space. This is for two reasons. Firstly, heaters are the archetypal "pure resistive load" so every heater draws 100 % *real power* and 0 % *reactive power*. Secondly, heaters all tend to draw close to this maximum power available at the socket (for the UK, this maximum is 13 amps × 230 volts = 3 kW).

For a recent review of NILM techniques, see [43].

Higher Harmonics and Electrical Noise

Many loads draw distorted currents due to their inherent physical characteristics (e.g. computers, photo copiers etc). Shaw *et al* [59] sampled current waveforms at 8,000 Hz and then computed spectral envelopes that summarize time-varying harmonic content. This provides further information which, combined with the traditional NILM approach, can be used to distinguish devices. This technique requires the use of specialised sampling hardware which is not readily available.

Froehlich *et al* [32] report disaggregation techniques which measure the very high frequency (10s or 100s of kHz) noise produced by many appliances. Some devices produce a specific noise signature at start up and some devices produce noise signatures during a continuous run (see figure 1.10). Devices like computer power supplies and TVs produce an especially large amount of electrical noise on the power line due, at least in part, to their "switch-mode" power supplies.

See [32] for a review of other "noise-based" disaggregation techniques.

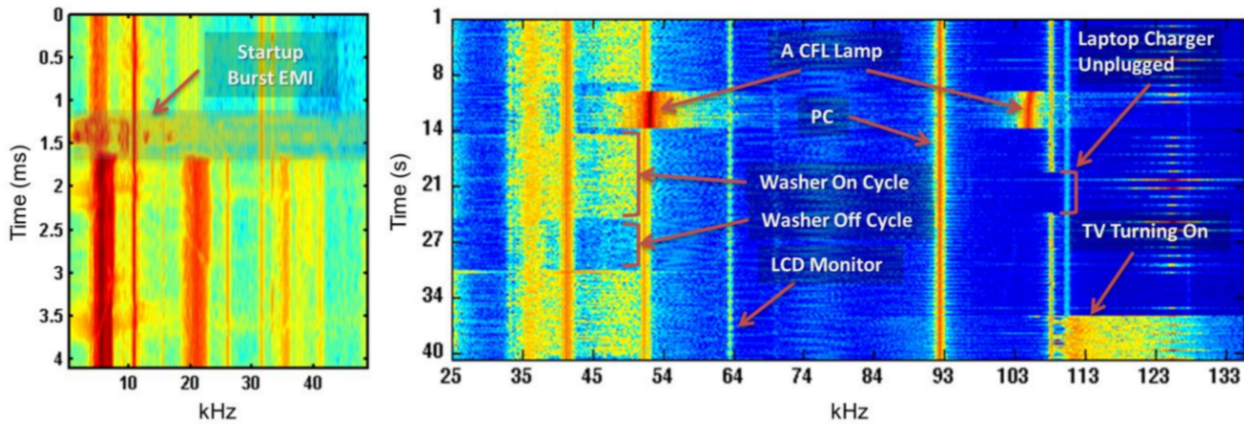


Figure 1.10.: Taken from Froehlich *et al* [32]. "*(left) Transient voltage noise signatures of a light switch being turned on. Colors indicate amplitude at each frequency. (right) Steady state continuous voltage noise signatures of various devices during various periods of operation.*"

Transient detection

Consider two different types of load: a washing machine's motor accelerating from standstill to full-speed and a large TV. Both loads might draw 200 watts when running. One difference between them is that the washing machine motor controller gently accelerates the motor over 30 seconds so the power consumption ramps from 0 watts to 200 watts over a 30 second period; whilst the TV instantly draws 200 watts. The washing machine's "transient" power ramp can be used as an identifying feature.

Most loads observed in the field have repeatable transient profiles [47]. Disaggregation based on recognition of transients permits near-real-time identification of devices. Transients in the aggregate data are identified by comparing them to a set of *exemplar* transients learnt during a training phase. Matching is performed using a least-mean-squares approach.

Sparse coding

Kolter, Batra & Ng [42] developed a novel extension to a machine learning technique known as *sparse coding* to disaggregate home energy monitor data with a temporal resolution of only one hour. Their method uses "structured prediction" to train sparse coding algorithms to maximise disaggregation performance.

This approach builds upon sparse coding methods developed for single-channel source separation. A sparse coding algorithm is used to learn a model of each device's power consumption over a typical week from a large corpus of training data. These learned models are combined to predict the power consumption of devices in previously unseen homes. Given the very low temporal resolution of the aggregate data, it is impressive that this technique achieves a test accuracy of 55 %.

Privacy concerns

George Hart, one of the developers of the "non-invasive load monitoring" technique, wrote an article in the June 1989 IEEE Technology and Society magazine [38], in which he discussed the privacy concerns about the NILM technique. He wrote:

> "A key feature of this new technique is its nonintrusive nature. The device can alternatively be installed on a utility pole at a distance from the site it is monitoring. With this mounting scheme, not even a momentary loss of electrical service is necessary for installation. From this unseen and unsuspected vantage point, the monitor has a view deep into the workings of the residence. After observing the residence for a short while, it generates a list of objects (appliances) and events (usages) that the occupants may consider completely private."

More recently, some consumers have have refused to allow utility companies to install smart meters in their home, due to concerns about privacy and electromagnetic interference [30].

The Netherlands passed a law in 2007 requiring 100 % roll-out of smart meters by 2013. However, this mandatory roll-out was suspended by parliament in July 2008 after increasing doubt about privacy, security and efficiency [10].

1.2. Broad aims and research direction

As discussed above, disaggregating energy data is not a new idea. Indeed, disaggregation algorithms have been successfully implemented commercially. However, these disaggregation algorithms usually require relatively sophisticated measurement hardware; hardware which is not currently readily available for domestic users. The ultimate aim of this current project is to develop a computational technique for disaggregating data from the most inexpensive and readily available form of smart meter: the home energy monitor.

One disadvantage of using home energy monitors is that these meters only provide a single scalar value, sampled at best once every six seconds and cannot measure the proportion of *real* versus *reactive* power; nor can they measure harmonics or distorted current waveforms. As such, the disaggregation techniques discussed above which rely on sophisticated measurement hardware are not directly applicable to this project.

The "sparse coding" technique discussed above (which does work with home energy monitor data) is not directly relevant either because that technique requires a large corpus of training data, which we don't have (although I did attempt to acquire such data).

The broad aim of this project is to produce an application which can deduce the start time, duration and energy consumption of individual device activations given an aggregate signal output by a home energy monitor. There does not appear to be an existing technique which is directly applicable to the current project. Instead, existing disaggregation techniques will need to be disassembled into their component parts and re-assembled into a system which can be applied to the current project.

1.3. Dissertation outline

chapter 2 describes the choice and configuration of the two power meters used to gather data for this project.

chapter 3 describes the prototype built to gain experience with the problem space. This prototype uses a "least mean squares" approach to disaggregation. This prototype performs as expected: it successfully locates a device signature within aggregate data but only if the device signature is recorded simultaneously with the aggregate data. The lessons learnt from this prototype are discussed.

chapter 4 describes a design for a fully functional disaggregation system. This system determines a set of "power states" for a device by first creating a histogram of the raw signature. After building and experimenting with an implementation of this design it became apparent that a better design was achievable; the new design is described in chapter chapter 6.

chapter 5 describes a framework built to ease the production of graphs from C++ using the command-line tool `gnuplot`.

chapter 6 describes the most recent design iteration. The design, implementation, testing and performance of this design is described in detail. This design successfully disaggregates three out of the four devices used during evaluation.

chapter 7 discusses conclusions, limitations, other applications of this work and further work.

1.4. Terminology used in this report

kilowatt hour (kWh) The kilowatt hour is a measure of electrical energy. The kWh is probably the most common unit for describing domestic energy consumption. When electrical utility companies refer to a “unit” of electricity, they mean the kWh. 1 kWh typically costs around 12.5 pence. The average electricity consumption per household in the UK is ∼9 kWh per day = 3,300 kWh per year. 1 kWh ≡ 3.6×10^6 Joules ≡ 860 dietary calories ≡ heat released by the combustion of ∼100g of coal.

Device signature This is the raw reading recorded by the meter which goes in between the device and the wall socket. Device signatures are used to *train* our disaggregation system. An example is shown in the bottom panel of figure 1.11.

Device fingerprint This is the device's “fingerprint” in the aggregate data. An example is shown in the top panel of figure 1.11. It is the result of the device's signature summed with an unknown number of unknown devices. The ultimate aim of this project is to confidently detect device fingerprints in noisy aggregate data.

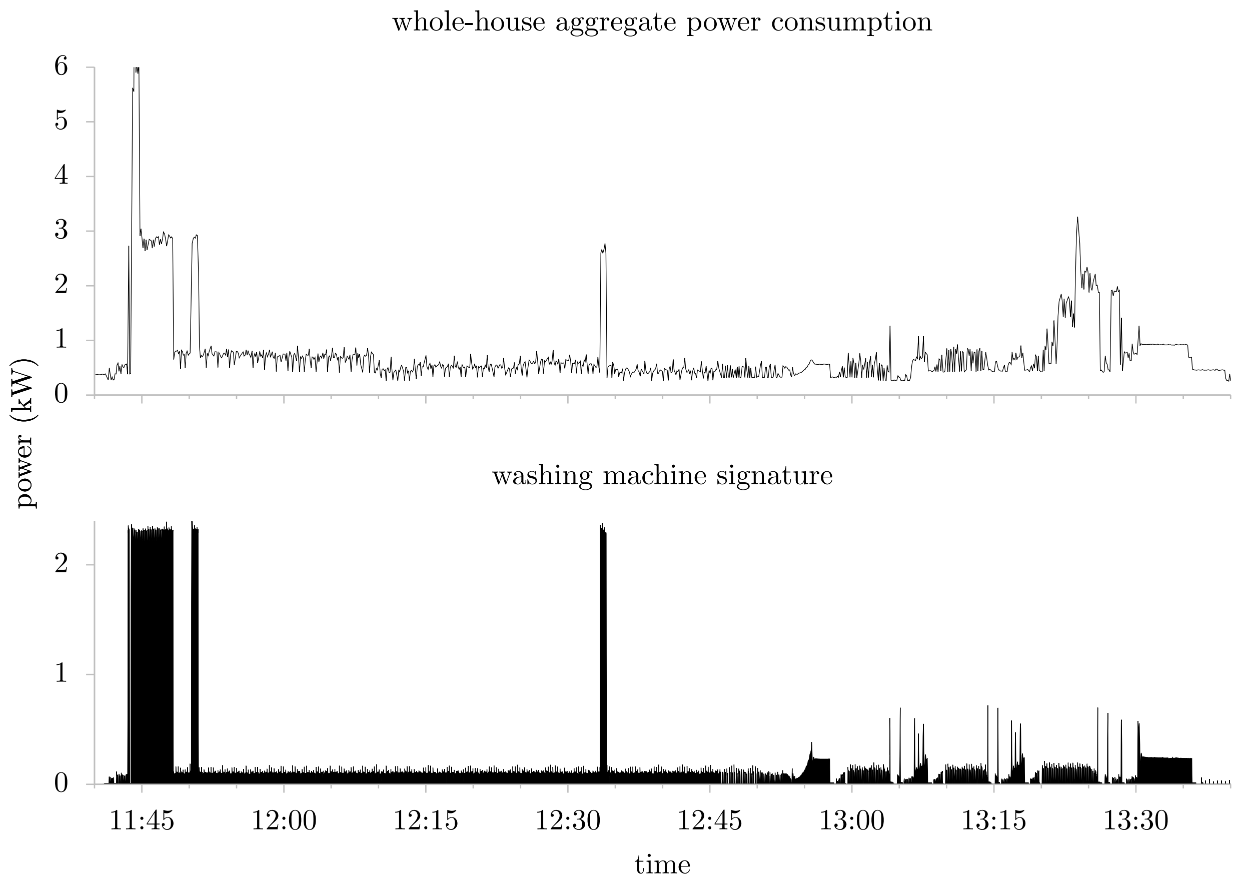


Figure 1.11.: The top panel shows whole-house aggregate power consumption. The bottom panel shows the power consumption of just the washing machine. The data in the two panels was recorded simultaneously. The effect of the washing machine on the aggregate power consumption is what we will call the “fingerprint” of the washing machine.

2. Setting up the measurement and logging equipment

This project requires two power meters: one for recording power consumption from a single device and one for recording whole-house aggregate power consumption.

2.1. Recording whole-house aggregate power consumption

There are several home energy monitors available for around £40-£80 (and some electricity utility companies give them away for free). Home energy monitors are primarily designed to *display* live energy consumption to the user but our project requires that the aggregate data be *logged* for subsequent analysis. Two different brands of meter were evaluated for use in the current project: an AlertMe (borrowed) and a Current Cost EnviR (my own).

Figure 2.1.: The Current Cost clamp installed in our home.

A unique feature of the AlertMe is that it does not have a hardware display[1]: instead it transmits data live to AlertMe's website. Users view their power consumption through AlertMe website. Software developers can access their own smart meter data through AlertMe's API[2]. Unfortunately the AlertMe did not function reliably (perhaps it was a faulty unit) so an alternative meter was acquired.

The Current Cost meter is popular with those who like to log energy consumption on a computer. For example, over 50 IBM staff members currently log their home energy consumption using Current Cost monitors[3]. It has a USB port which allows easy access to live data from a computer. The Current Cost has two main components: a clamp sensor (figure 2.1) and a display unit. The sensor sends data wirelessly to the display. The display has a basic internal logging facility but, crucially, it does not store high temporal resolution data. This project requires the highest temporal resolution data available from the Current Cost (1 sample every 6 seconds) which is *produced* by the Current Cost but is not *logged*. So a laptop was dug out from the loft to take responsibility for reading live data from the Current Cost. A simple perl logging script was written[4] to log two parameters: UNIX timestamp and power (in watts). The end result is a log of aggregate energy consumption with a temporal resolution of 1 sample every 6 seconds and a measurement resolution of 1 watt. An example aggregate data plot is shown in figure 2.2.

The reliability of the wireless connection between the sensor and the display proved to decrease as the distance between the two units increased so the sensor, display and laptop were located together, next to the fuse box.

[1] The AlertMe I borrowed did not have a display device but, within the last month, AlertMe started selling a display device.

[2] alertme.com/business/platform.html

[3] realtime.ngi.ibm.com/currentcost

[4] My Perl script is on github at github.com/JackKelly/CurrentCostLogger. It is based on Paul Mutton's excellent tutorial available at jibble.org/currentcost with modifications from Jamie Bennett's blog available at linuxuk.org/2008/12/currentcost-and-ubuntu

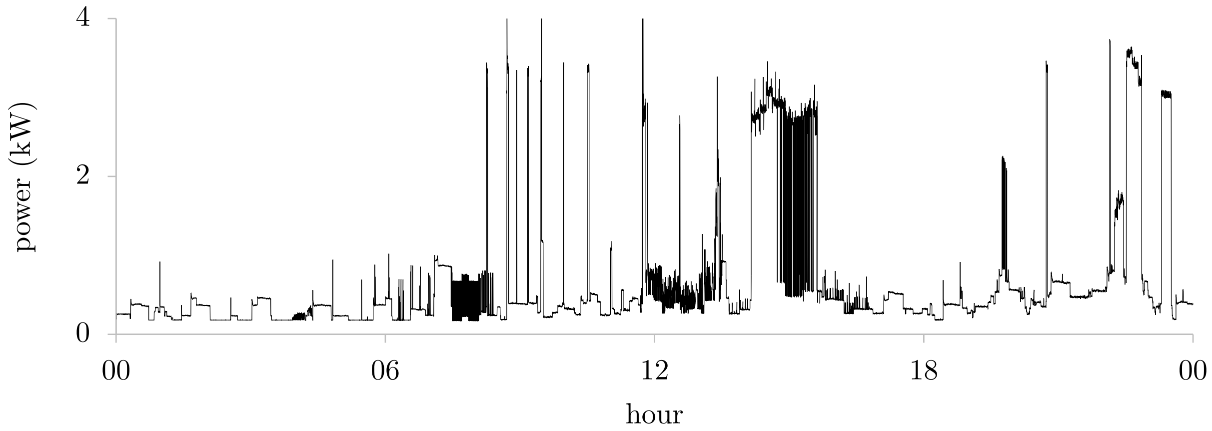


Figure 2.2.: Example aggregate data recorded over a 24-hour period by the Current Cost.

2.2. Recording device signatures

The single-appliance meter used for this project is a "WattsUp" meter (which was kindly lent to me by my supervisor, Dr Knottenbelt). This plugs into the wall socket. The device under test plugs into the WattsUp. The WattsUp records a sample once a second for up to 6 hours (it has 2 MBytes of internal memory). The WattsUp log includes the following information:

- power (in steps of 0.1 Watts)
- current
- voltage
- power factor

Data are downloaded from the WattsUp to a computer via USB using a Windows utility which outputs a CSV file. The CSV file is loaded into LibreOffice Calc to remove all columns except the "watts" column. This single-column data is now ready for use. An example signature recorded by the WattsUp is shown in figure 2.4.

Figure 2.3.: The WattsUp monitor. (Photo taken from lakerenergymatters.blogspot.com)

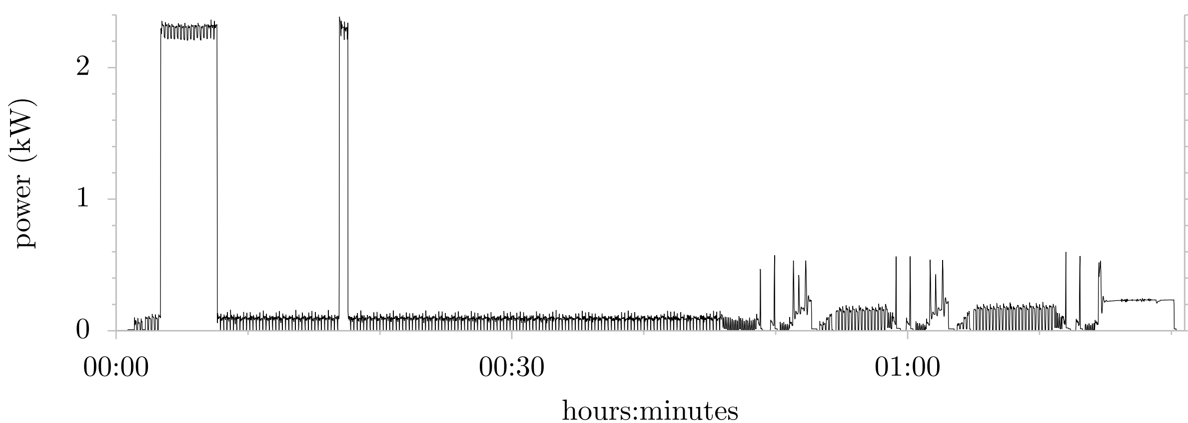


Figure 2.4.: Example washing machine signature recorded by the WattsUp. The 2.3 kW spikes are the water heater. The 200 watt section at 1:15 is the fast spin.

3. Early prototype

In order to get a feel for the task, we first set a goal to build a deliberately over-simplified prototype designed to automatically align data recorded simultaneously by the WattsUp and the Current Cost. The challenge is illustrated in figure 3.1.

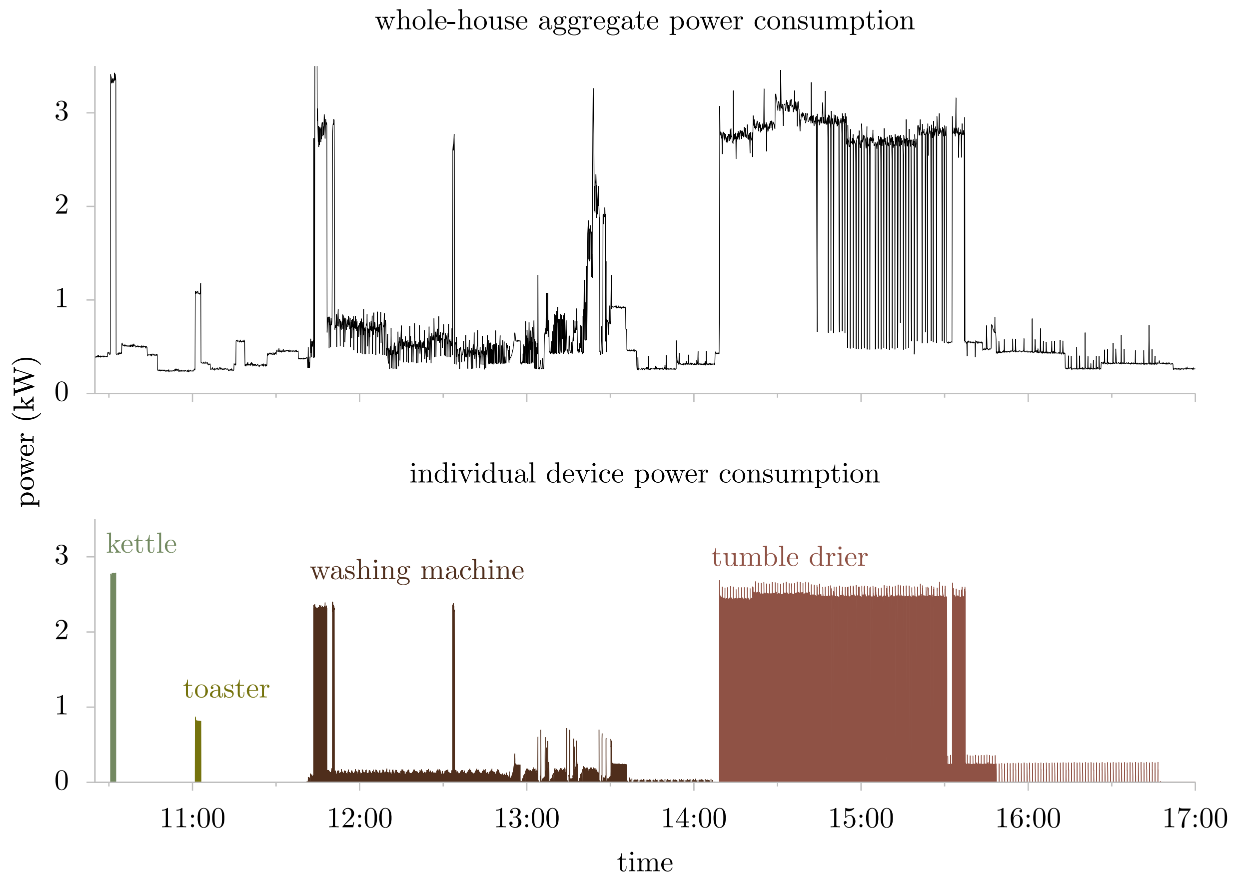


Figure 3.1.: Aggregate data aligned with individual device signatures. The top panel shows whole-house aggregate power consumption recorded by the Current Cost. The lower panel shows the individual power consumption of four devices. The data in the top and bottom panels were recorded simultaneously.

3.1. Prototype version 1

The algorithm takes an unsophisticated, brute-force approach to the disaggregation problem. Starting at the beginning of the aggregate data, we calculate a mean-squared error value for the match between the device signature and the first stretch of aggregate data. Then shift one step forward in the aggregate data and calculate a new mean-squared error value. Continue until we get to the end of the aggregate data. Finally, report the position which produced the smallest error value (see algorithm 3.1). This algorithm was tested by creating a cropped copy of a device signature and then asking the algorithm to locate the cropped signature within the original, uncropped signature (which worked perfectly).

The algorithm was then tested against the data used to produce figure 3.1 (i.e. 24 hours of aggregate data plus device signatures recorded in that same 24 hour period for the kettle, toaster,

Algorithm 3.1 Using Least Mean Squares to locate *signature* within *aggregateData*. Recall that the aggregate data is sampled once every 6 seconds whilst the signature data is sampled once a second; hence the "6" in the *meanSquaredError* calculation.

LOCATE(*aggregateData, signature*) Returns index in *aggregateData* at which *signature* starts
  Let a_i be the ith sample in the *aggregateData*
  Let s_i be the ith sample in the *signature*
  Let $minErrors \leftarrow infinity$
  Let $location \leftarrow -1$
  for all $a_i \in aggregateData$ **do**
    $meanSquaredError \leftarrow \left(\sum_{j=0}^{signatureLength \div 6} (a_{i+j} - s_{j \times 6})^2\right) \div (signatureLength \div 6)$
    if $meanSquaredError < minError$ **then**
      $minError \leftarrow meanSquaredError$
      $location \leftarrow i$
    end if
  end for
  return *location*

washing machine and tumble dryer). This first prototype gave nonsense answers for every device!

3.2. Prototype v2: compensating for dropped measurements

The Current Cost home energy monitor occasionally fails to detect a reading sent over the air from the clamp meter. Perhaps the algorithm described above failed because it assumes that every sample in the aggregate data has been recorded exactly 6 seconds apart. If this assumption is false then even if the beginning of the WattsUp data is aligned perfectly with the Current Cost data, the two will drift out of sync because there are missing samples in the Current Cost data. Examining the data logs it appears that the Current Cost drops 10 % of the readings sent to it and that not all readings are taken exactly 6 seconds apart.

Prototype version 2 no longer assumes that consecutive aggregate data samples are exactly 6 seconds apart. Instead, it reads the timestamp of each aggregate data sample and compares this to the correct WattsUp sample. For example, if the aggregate data contains consecutive samples taken at 0, 6, 13 and 25 seconds then version 2 of the algorithm compares these with WattsUp samples 0, 6, 13 and 25, whereas version 1 of the algorithm would have used WattsUp samples 0, 6, 12 and 18.

Version 2 located the tumble drier perfectly but still gave nonsense answers when asked to locate the washing machine, toaster or kettle.

3.3. Prototype v3: offsetting & compensating for sample-rate mismatch

3.3.1. Offsetting

A typical home continues to consume several hundred watts even when the occupants believe that "everything is off". This *"vampire power"* can largely be attributed to devices which continue to draw power in standby mode and by devices which are permanently on (the alarm system, cordless phone base stations, the fridge etc.). Previous versions of the prototype compare the WattsUp device signature against an aggregate signal which is always several hundred watts above zero, and this baseline moves up and down as devices around the house change state. The prototype was modified in an attempt to compensate for this moving baseline. When it starts calculating the mean-squared error at each position in the aggregate data, the new version starts by calculating the mean of the first 50 samples of the aggregate data and subsequently using this mean as an

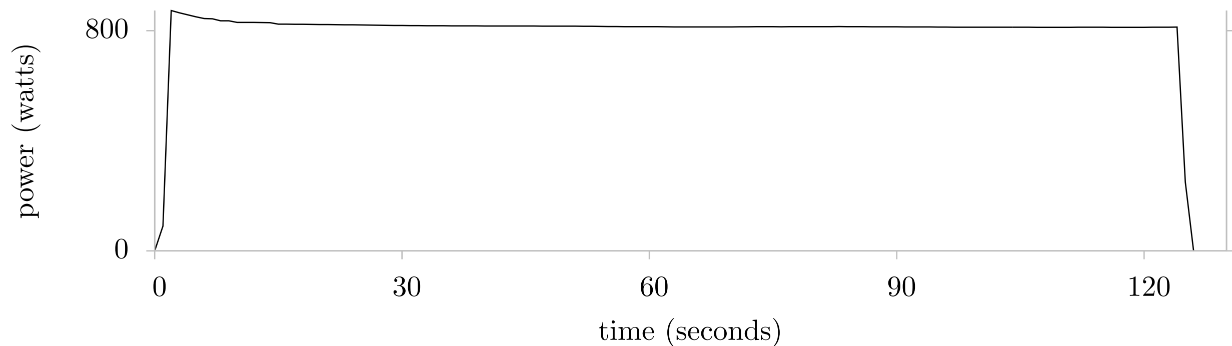


Figure 3.2.: Toaster raw signature with only a single zero at the start and end

"offset". The algorithm takes this offset into account when calculating the mean-squared for the whole WattsUp signature.

3.3.2. Sample-rate mismatch

The WattsUp records a sample exactly once per second but the Current Cost records an instantaneous sample roughly every 6 seconds. Previous versions of the prototype handled this mismatch by simply throwing away 5 out of every 6 WattsUp samples. This is fine for events which have a period of 12 seconds or longer but the washing machine and tumble drier signatures contain features which are considerably shorter than 12 seconds. The prototype was modified to better utilise all the information available in the WattsUp signature by comparing every Current Cost sample to the 6 nearest WattsUp samples.

3.3.3. Accuracy of prototype v3

Prototype 3 performs no more accurately than prototype 2 (i.e. it reliably locates the tumble drier but still produces garbage answers for the washing machine, toaster and kettle). Prototype 3 also runs 6 times slower than prototype 2!

3.4. Further experiments: finally achieving automated alignment of all devices

3.4.1. Kettle and toaster

The code automatically removes leading and trailing zeros from each signature so the signature after preprocessing has only a single zero at the front and back. This means that the toaster signature starts immediately with a rapid transition lasting 2 seconds from 0 to 800 watts and ends with a rapid transition from 800 to 0 watts (figure 3.2). It would be all to easy for the algorithm to miss the start and end transitions for the toaster and kettle, especially given the sample rate mismatch between the WattsUp and Current Cost. So 50 zeros were added to the start and end of each signature and then the prototype was able to successfully locate the kettle and toaster! But how can we get a lock on the washing machine fingerprint?

3.4.2. Washing machine

Looking more closely at the aggregate data during the time when the washing machine was on, it is clear that several unknown, power-hungry devices change state whilst the washing machine is running (figure 3.3). For example, at around 11:44 a ∼3 kW device turns on for a minute; at around 12:10 a ∼200 W device turns off (the fridge's compressor, maybe?); and at around 13:20, a ∼1 kW device turns on. This "background noise" means that when the algorithm stumbles upon the correct alignment it will calculate an unfavourable mean-squared error value because the

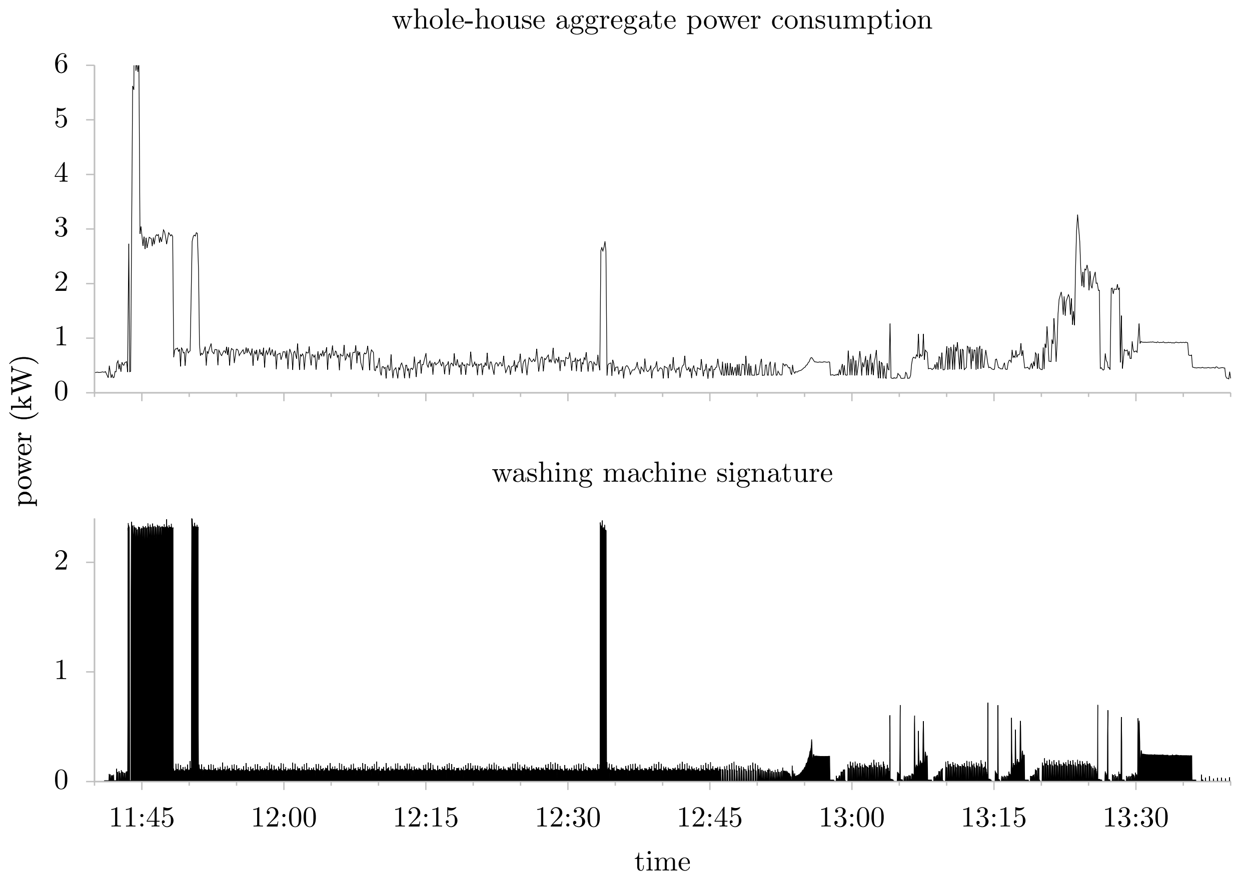


Figure 3.3.: Zoomed into washing machine aligned with aggregate data. The top panel shows the whole-house aggregate power consumption. The bottom panel shows the washing machine signature. The data for the two panels was recorded simultaneously.

algorithm assumes that the baseline does not change over the course of the device's run. To put this another way: the least mean squares algorithm will only calculate a favourable error score over short stretches of the washing machine signature, not over the entire 2 hour run. (This issue is discussed further in section 6.1).

To test this hypothesis, the washing machine signature was cropped to just the data from 12:30 to 13:00 (because there do not appear to be any large, unknown devices changing state during this period in the aggregate data). The algorithm correctly located this cropped signature in the aggregate data.

3.5. Failure to generalise

So, the least mean squares algorithm can successfully find all four devices in the aggregate data. Does this mean that the project is finished? Does this least mean squares algorithm successfully find all instances of the washing machine in the aggregate data from only a single training example?

To find out, a second washing machine signature was recorded and the prototype attempted to find this new signature within the old aggregate signal (i.e. the signature and the aggregate signal were recorded at separate times). The system failed to locate the new signature, even when cropped.

Thus, as expected, this simplistic approach works only if the aggregate data and signature data are recorded simultaneously. A more sophisticated approach will be required in order to locate device fingerprints in aggregate data recorded at a different time to the signature.

3.6. Prototype summary: lessons and limitations

1. It is necessary to explicitly take into consideration the fact that the Current Cost drops 10 % of its samples.
2. A device is very unlikely to produce exactly the same signature twice. Hence trying to locate *waveforms* is unlikely to succeed.
3. It probably makes more sense to think in terms of locating *state transitions* rather than locating entire waveforms.

The main limitations of the prototype stem from the over-simplistic assumptions; namely:

1. The prototype assumes that the signatures were recorded at the same time as the aggregate data. Whilst this was an interesting experiment, this assumption is clearly not viable for a production system.

4. First design: histograms and power states

4.1. Aims

The user will train the system to recognise a device by providing a signature file for that device. During disaggregation, the system must locate every occurrence of that device within an aggregate signal. The aggregate signal will be recorded at a different time to the signature. Training may require limited human intervention but disaggregation should be completely automated.

4.2. Broad design principals

We learnt in the previous chapter than we cannot rely on simple pattern-matching of the signature waveform against the aggregate signal so we need to abstract from the signature to a representation which is sufficiently general to allow for routine variation in device behaviour but also sufficiently descriptive to allow us to discriminate different appliances. This representation should be loosely based on the physical characteristics of the world we're attempting to model.

4.3. Which programming language?

This project requires a language which is efficient, object orientated and has a large existing library of signal processing tools. This set of requirements pointed towards C++ with help from the Standard Template Library and the Boost library.

Matlab was considered as a prototyping platform but was dismissed because, frankly, I wanted this project to provide an intensive C++ education (which it certainly did).

4.4. Training strategy

The strategy, per appliance, is:

1. Record a raw log of the energy consumption of each device using the WattsUp meter.
2. Extract and store two abstractions from the raw log (this should be automated as much as possible but may require human intervention at some stages):
   a) Let's assume that a device's power consumption tells us something useful about the internal state of the device. The first task is to determine a set of "*power states*". For example, a kettle would have two "power states": *off* (0 W) and *on* (2.7 kW). On the other hand, a washing machine is likely to have many power states like *water heater on* (2.3 kW), *slow spin* (100 W), *fast spin* (300 W) and *anti-crease* (alternating between 0 W and 50 W).
   b) Once the set of "power states" are established, determine the *sequence* in which the power states occur in this raw power log. For a kettle, this sequence would simply be "power state *on* lasts for 60 seconds". For a washing machine, this sequence might be "first *spin gently* for 5 minutes, then *heat water* for 1 minute, then *spin gently* for an hour, then *spin quickly*." Locate any repeating sub-sections of the sequence and encode the range of permissible repeats. Encode the likelihood, given previous states, of a state change happening at a particular time (represented perhaps as a probability density function). For example, the fridge's compressor continuously cycles between *on* for about 5 minutes and *off* for about 10 minutes. We know it's extremely unlikely for

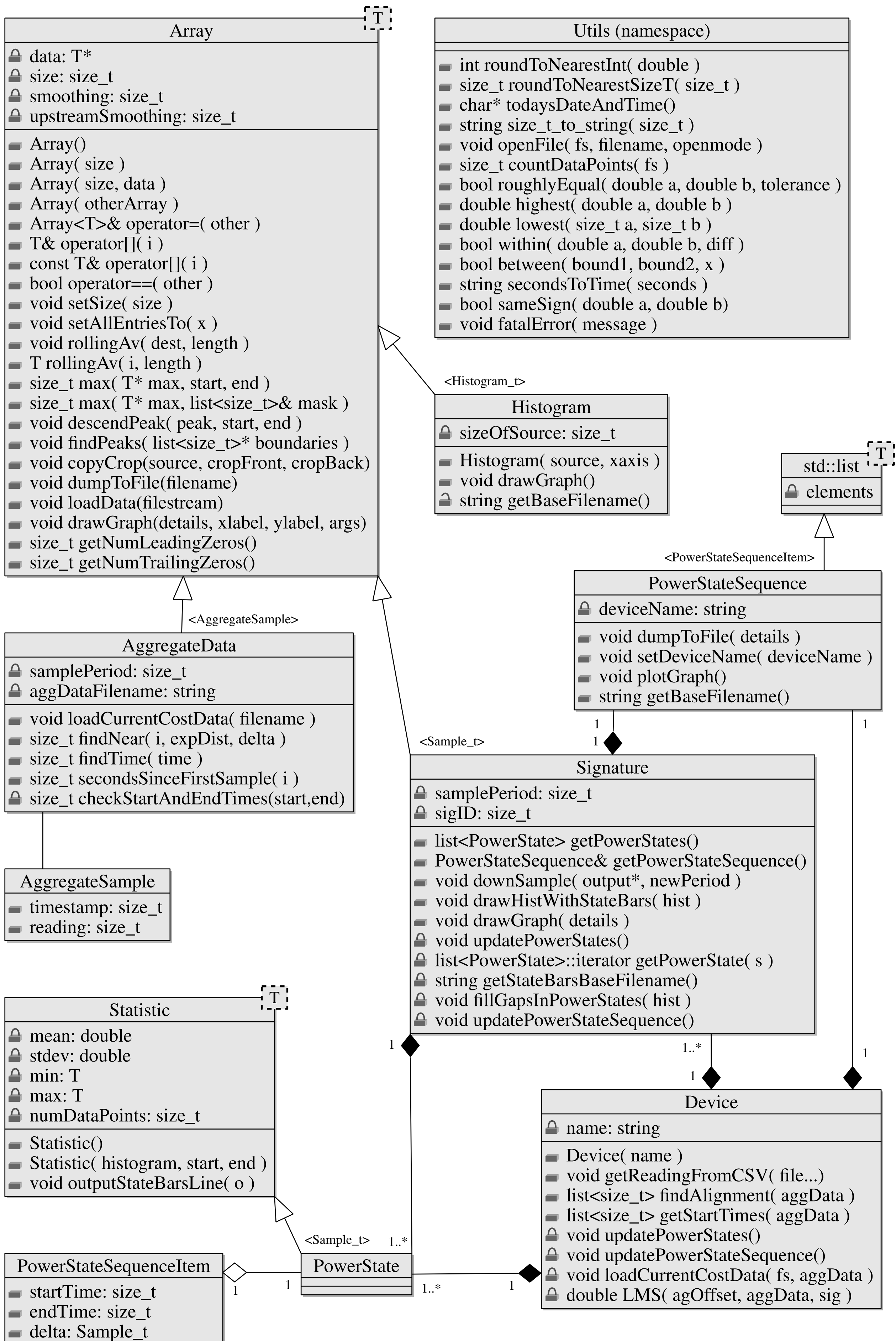


Figure 4.1.: UML Diagram. `Sample_t` and `Histogram_t` are both typedefs for a `double` (typedefs were used to allow other basic types to be used as the “sample type” and “histogram type” if required). `Utils` is not a class, it is a namespace containing useful utilities which do not belong in specific classes.

the compressor to come on only 10 seconds after turning off but we also know that it may come on after only 5 minutes if the fridge has just been filled with room-temperature food.

3. We'll use the *transitions between states* to identify each device in the aggregate signal so we need to score each power state transition for its expected usefulness as an identifying characteristic. For example, a transition from a 20 W state lasting 1 second to a 25 W state lasting 2 seconds will be fairly useless as an identifying characteristic because it will easily be missed. Far better would be a transition from a 100 W state lasting 1 minute to a 2000 W state lasting 5 minutes. Basically: we're looking for state transitions which will *stand out* from the noise in the aggregate data.

4.5. Disaggregation strategy

For each device (using a washing machine as a concrete example):

Select the top scored *transition between states.* For example, the top scored *transition between states* for the washing machine might be the transition from *slow spin* at 100 W to *heat water* at 2.7 kW. Search through the aggregate data for a 2.6 kW jump. When a 2.6 kW jump has been found, return to the *power state sequence* and select the second highest ranked transition between states and try to find it in the aggregate data, whilst taking into consideration the expected length of time between the highest ranked transition between states and the second highest ranked. For the washing machine, the second highest ranked state transition might be the transition from *heat water* back to *slow spin*, which is expected about 5 minutes after the transition from *slow spin* to *heat water.* Continue searching for salient transitions between states until we can be confident beyond reasonable doubt that we have located the device in question.

4.6. Implementation

4.6.1. Doxygen HTML code documentation

Before we discuss the implementation details, please note that the code documentation is available at www.doc.ic.ac.uk/∼dk3810/disaggregate. This HTML documentation was automatically generated from the code using the documentation tool Doxygen[1]. The HTML documentation lists all the classes and, for each class, lists each member function and member variable, along with many of the comments from the code. Some information is presented graphically including class inheritance diagrams, collaboration diagrams, caller and call graphs.

4.6.2. Extracting a set of power states from the raw signature

The following strategy has been implemented to extract a set of *power states* from a raw signature (figure 4.2 shows an example output from the code): first we create a histogram of the raw signature by passing a `Signature` object to the `Histogram` constructor, then we identify the largest peaks in the histogram using `findPeaks()` and then we record a set of statistics for each of these peaks (min, mean, max, stdev) using the `Statistic` constructor.

The `Array::findPeaks()` function

`findPeaks()` attempts to find "peaks" in a histogram (see figure 4.2). The basic strategy can be visualised as a blind but intrepid hill-climber travelling across an uncharted mountain range: at every point the walker can feel in her legs whether she is ascending, travelling along a flat or descending. She can infer that she has recently passed a peak if she ascends, then travels along a flat and then descends.

[1] doxygen.org

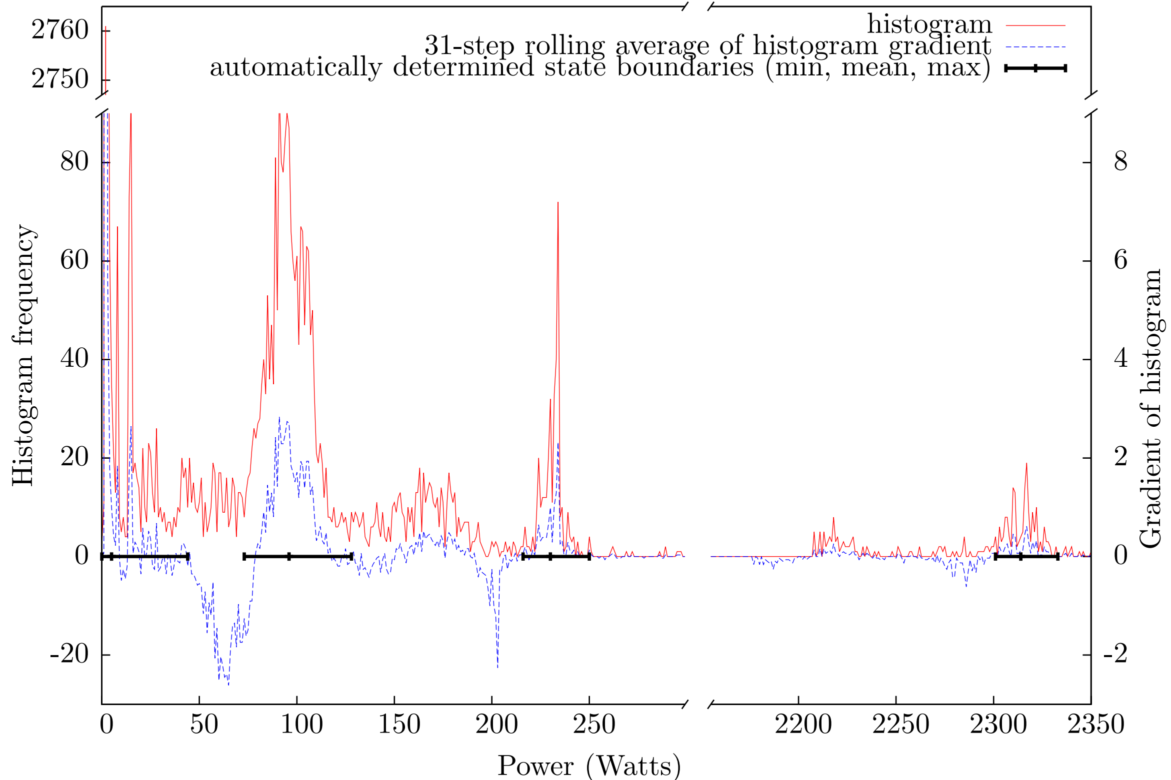


Figure 4.2.: Automatically determined power states from a signature histogram for a washing machine. The red solid line shows a histogram of washing machine data (unsmoothed). Automatically determined state boundaries are shown in black. The state at around 100 W is the slow spin during the main washing cycle; the state around 240 W is the fast spin at the end of the cycle and the state at 2310 W is the water heater (the washer only has a cold water input so it must heat the water itself). After creating the histogram, the `findPeaks()` function works through the histogram from the furthest right. It determines the gradient of each point on the histogram, takes a rolling average of this gradient (shown in dashed blue) and then uses this smoothed gradient to automatically find state boundaries.

In a similar fashion, `findPeaks()` "senses" the gradient underfoot by calculating the gradient at every point on the histogram and then smoothing this gradient. `findPeaks()` works backwards through the histogram (i.e. starts at the furthest right and travels left) because the algorithm assumes that we always start on a flat gradient. We can be confident that the very end of the histogram will always have a frequency of zero because the furthest right of the histogram records the frequency of samples with a value of 3,500 Watts, but no domestic device can draw above 3,000 Watts.

The code labels the flat sections between peaks as "no mans' land". The constant `KNEE_GRAD_THRESHOLD` defines the gradient threshold which delineates the transition from "no mans' land" to ascent (i.e. the "knee" of the ascent curve); and the constant `SHOULDER_GRAD_THRESHOLD` defines the gradient threshold which delineates the transition from ascent to a flat gradient (i.e. the "shoulder" of the ascent curve). The function `findPeaks()` returns a list of indices bounding the start and finish of each peak.

The C++ code for `findPeaks()` follows:

```
list Array::findPeaks() {
  list<size_t> boundaries; // what we return
  const double KNEE_GRAD_THRESHOLD = 0.00014;
  const double SHOULDER_GRAD_THRESHOLD = 0.00014;
  enum { NO_MANS_LAND, ASCENDING, PEAK, DESCENDING, UNSURE } state;
```

```
state = NO_MANS_LAND;
double kneeHeight=0, peakHeight=0, descent=0, ascent=0;
// Construct array of gradients
Array<Sample_t> gradient;
getDelta( &gradient, -1 );
// Construct an array of smoothed gradients
Array<Sample_t> smoothedGrad; // return parameter for rollingAv
gradient.rollingAv( &smoothedGrad );
// start at the end of the array, working backwards.
for (size_t i=(size-HIST_GRADIENT_RA_LENGTH); i>0; i--) {
  switch (state) {
    case NO_MANS_LAND:
      if (smoothedGrad[i] > KNEE_GRAD_THRESHOLD) {
        /* when the gradient goes over a certain threshold,
         * mark that as ASCENDING,
         * and record index in 'boundaries' and kneeHeight */
        state=ASCENDING;
        boundaries.push_front( i );
        kneeHeight = data[ i ];
      }
      break;
    case ASCENDING:
      // when gradient is around 0 then state = PEAK and record peakHeight
      if (smoothedGrad[i] < SHOULDER_GRAD_THRESHOLD) {
        state=PEAK;
        peakHeight = data[ i ];
        ascent = peakHeight - kneeHeight;
      }
      break;
    case PEAK:
      // when gradient goes below a certain threshold, state = DESCENDING
      if (smoothedGrad[i] < -SHOULDER_GRAD_THRESHOLD) {
        state=DESCENDING;
      }
      if (data[i] == 0) {
        state=NO_MANS_LAND;
        boundaries.push_front( i );
      }
      break;
    case DESCENDING:
      /* when gradient goes to 0, check height.
       * If we've descended less than 30% our ascent height then
       *   don't mark in 'boundaries', instead re-mark as ASCENDING
       * else mark in 'boundaries' and state=NO_MANS_LAND */
      if (smoothedGrad[i] > -KNEE_GRAD_THRESHOLD || data[i]==0) {
        // check how far we've descended from the shoulder
        descent = peakHeight - data[ i ];
        if (descent < (0.3 * ascent)) {
          state=UNSURE;
        } else {
          state=NO_MANS_LAND;
          boundaries.push_front( i );
        }
```

```
            }
            break;
        case UNSURE:
            // We were descending but hit a plateau prematurely.
            // Find out if we're descending or ascending
            if (smoothedGrad[i] < -SHOULDER_GRAD_THRESHOLD) {
                state = DESCENDING;
            } else if (smoothedGrad[i] > KNEE_GRAD_THRESHOLD) {
                state = ASCENDING;
            } else if (data[i]==0) {
                state=NO_MANS_LAND;
                boundaries.push_front( i );
            }
            break;
        }; // end switch
    }    // end for
    if (state != NO_MANS_LAND) {
        boundaries.push_front(0);
    }
    return boundaires;
}
```

The `Array::rollingAv()` function

The `Array::rollingAv( Array* output, const size_t length )` function smooths an `Array` object by taking a rolling average of specified `length` (figure 4.3). Parameter "`output`" is the returned smoothed Array object.

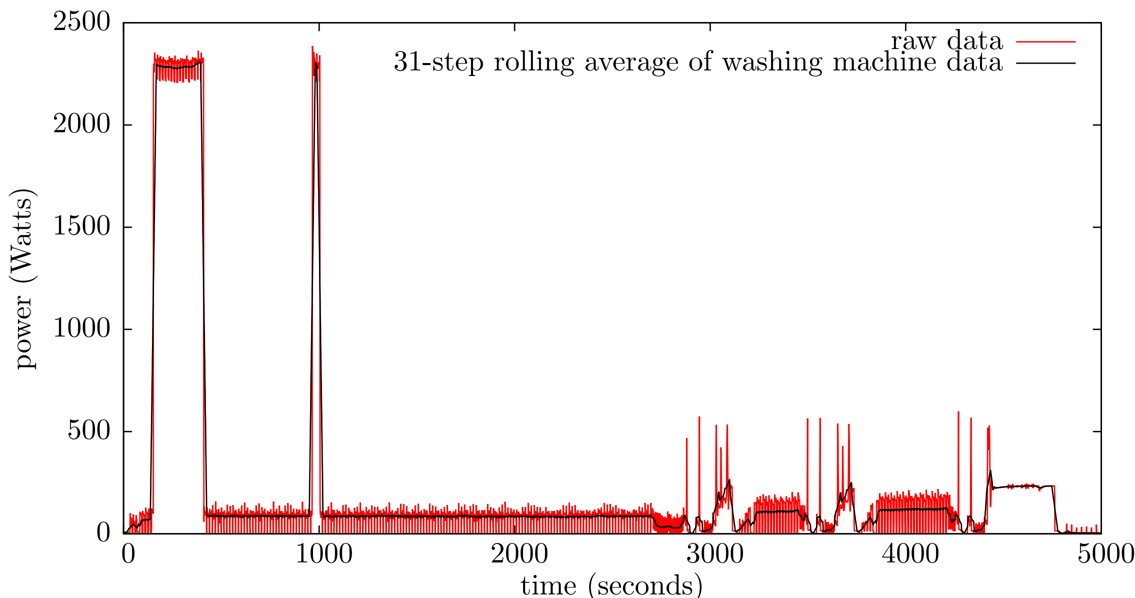


Figure 4.3.: Raw and smoothed data

Why does `rollingAv()` return its output in the form of an `Array* output` *parameter* instead of *returning* an `Array` object? Consider an alternative function which returns an `Array` object. The function would start by creating a local `Array` object, populating this object and then returning a *copy* of this object. It is this inefficient copying that we wish to avoid. So we let `rollingAv`'s caller statically declare an `Array` object and pass this object to `rollingAv()` to populate it. We pass in an `Array*` instead of an `Array&` purely as a cosmetic style convention suggested by Bjarne Stroustrup [63] so that `rollingAv` is always called like this: "`array.rollingAv( &rollingav )`";

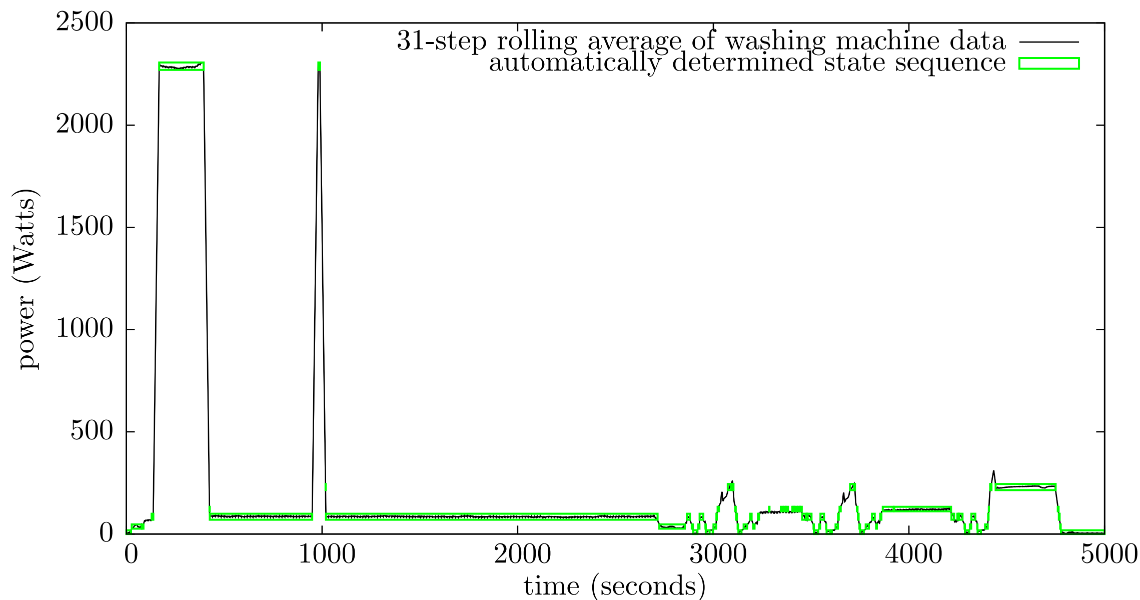


Figure 4.4.: Automatically determined power state sequence for a washing machine (please note that these power states do not correspond to those in figure 4.2) The height of the state boxes indicates the min and max values for that power state.

the ampersand reminds us that `rollingav` will be modified. A third option would have been to dynamically create a `new Array` object at the start of the `rollingAv` function and to return a pointer to this object. The problem with this strategy is that this shifts responsibility for deleting the object onto the caller function which is risky[2].

I experimented with smoothing the raw signature prior to creating a histogram but this approach is unlikely to be statistically valid and also doesn't appear to improve performance. The histogram is not binned. Binning might be an alternative to smoothing.

4.6.3. Extracting a power state sequence

A simple strategy to extract a *power state sequence* has been implemented. The code simply runs through the raw signature and, for each sample, tests whether this sample falls within the bounds of any *power state*. An example output of my code is given in figure 4.4. At the moment, the "power state bounds" are defined as the minimum and maximums, although it may be better to use one stdev above and below the mean.

4.6.4. Data output

The "histogram" design can be invoked at the command line by supplying `disaggregate` with the `--histogram` switch (see the software user guide in appendix A for more details). For example the command "`./disaggregate --histogram -s washer.csv -n "washer"`" will create a histogram for the washer signature, infer a set of power states and infer a power state sequence. The following graphs are output to the `data/output` path:

washer-afterCropping.svg The raw signature file.

washer-hist-UpstreamSmoothing1.svg The histogram. (The "UpstreamSmoothing1" simply says that the signature was not smoothed prior to creating the histogram)

[2]This paragraph describes my thought process while I was writing this code over a month ago. I now know the situation is more nuanced for at least two reasons: firstly, perhaps a `std::auto_ptr` could have been used to return a pointer to a dynamically assigned `Array` object (thereby implementing the *Resource Acquisition Is Initialisation* (RAII) programming idiom). Secondly, both GCC and MSVC compilers implement "return value optimisation" hence code which returns a copy of a temporary object should be optimised such that the temporary object is simply returned, rather than copied, especially in `C++0x` [23].

washer-histWithStateBars.svg The histogram overlayed with the histogram gradient and the inferred power states (similar to the graph show in figure 4.2).

washer-powerStateSequence.svg The power state sequence (similar to the graph shown in figure 4.4).

4.7. Flaws in the "histogram" design

After implementing the code to determine a set of power states and a power state sequence, it became apparent that this design approach was not optimal for a number of reasons. One reason was that the behaviour of the function `Array::findPeaks` (which aims to find "peaks" in signature histograms) changed dramatically if the constants `KNEE_GRAD_THRESHOLD` and `SHOULDER_GRAD_THERSHOLD` were altered by small amounts. Even more worryingly, parameters which work well for one signature do not work well for other signatures. This made the code feel very "fragile". Other serious flaws of the "histogram" design approach will be discussed in chapter 6 which describes an alternative and superior design.

5. gnuplot template instantiation system

This project is, at its heart, a signal processing project. As development progressed, it became clear that an essential requirement for efficient development is to be able to visualise data as it flows through the code. In the early stages of the project, data were output from the code as text files and plotted using Matlab or LibreOffice Calc[1]. It quickly became apparent that this relatively labour intensive approach to plotting was a development bottleneck and so a new plotting system was required with the following characteristics:

1. Fully automated generation of plots directly from the code.
2. Customisable plots
3. Easy integration with C++

`gnuplot`[2] is a free, open source command-line graphing utility. It is tremendously powerful and offers a lot of control. `gnuplot` takes two basic forms of input: a script file and (at least one) data file. For example, a simple script to plot a device signature as an `svg` image file might look like this:

```
set terminal svg size 1200 800; set samples 1001
set output "data/output/washer.svg"
set title "washer signature"
set xlabel "time (seconds)"
set ylabel "power (watts)"
plot "washer.dat" with lines linewidth 1
```

`gnuplot` certainly has the capacity to draw every type of plot this project requires (and all the plots in this report were produced with `gnuplot`). It is trivial to design the C++ code to output numeric data to a text file. But how can the code automatically produce bespoke `gnuplot` scripts and call `gnuplot` to render the plot?

The first solution was to simply use the `popen()` UNIX system call. `popen()` initiates a pipe stream to an executable and allows that stream to be written to like a normal file:

```
FILE* gnuplotpipe = popen("gnuplot", "w");
// gnuplot is now running and ready to accept commands. We
// send commands to 'gnuplotpipe' as if it were a normal FILE*
fprintf(gnuplotpipe, "plot \"washer.dat\" with lines linewidth 3\n"
                     "exit\n");
pclose(gnuplotpipe);
```

This was fine for simple gnuplot scripts but became unmanageable for larger scripts (some `gnuplot` scripts in this project are over 170 lines long; far too much to hard-code into a C++ file).

The solution was to build a simple *template instantiation system*. A template file might look like this:

```
set title "TITLE"
set xlabel "XLABEL"
set ylabel "YLABEL"
plot "DATAFILE" with lines linewidth 1 title "DATAKEY"
```

[1] libreoffice.org/features/calc
[2] gnuplot.info

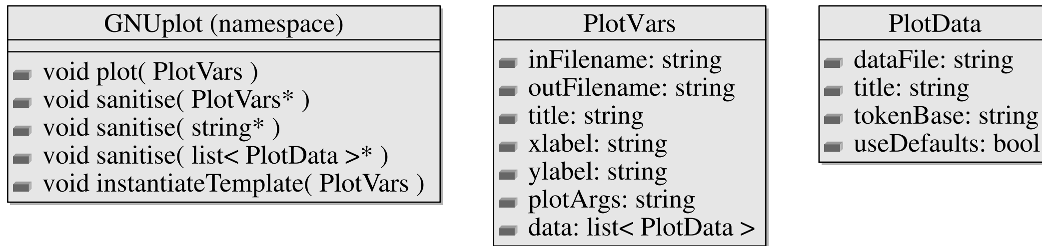


Figure 5.1.: The `GNUplot` namespace. `GNUplot::PlotVars` and `GNUplot::PlotData` are structs.

The template instantiation system replaces the *tokens* `TITLE, XLABEL, YLABEL, DATAFILE` and `DATAKEY` in the template file with whatever string might be appropriate for the current graph. The instantiated template is then sent to `gnuplot`. This template system has several attractive features:

- maximises reuse of `gnuplot` code
- `gnuplot` templates can be debugged more easily than hard-coded `gnuplot` scripts
- `gnuplot` commands can be altered by the user without having to recompile the C++ code

5.1. Implementation

The gnuplot template instantiation code lives in a separate "`GNUplot`" namespace (see figure 5.1) instead of a class because the gnuplot code does not need to store any state information. Template tokens are replaced by the UNIX text-processing utility `sed` (which is called from C++ using the `system`() function).

If a signal processing function wishes to produce a plot then it first must create a `GNUplot::PlotVars` struct containing the relevant details (see figure 5.1). This struct is then passed by reference to `GNUplot::plot()`. The first thing `plot`() does is to "sanitise" the strings by replacing any characters which may confuse either `sed` or `gnuplot`. `plot`() then instantiates a gnuplot template using `instantiateTemplate`().

Template files are stored in the `config/` directory. For maximum flexibility, there are two possible templates for each plot: a general template and a template specific to that device. This is useful because, for example, plotting a histogram for a *washer* may require broken axes whilst a histogram for a *kettle* may not; hence it is useful to allow a specific template to be specified for washer histograms. Consider a situation which requires a plot of a histogram for a washer device. `instantiateTemplate`() first looks for a template with the name "`washer-histogram.template.gnu`". If (and only if) this file is not found then the template file "`histogram.template.gnu`" is used instead.

5.1.1. `gnuplot` output

`gnuplot's` output format is configured using the "`set terminal`" command. For example, if we want `gnuplot` to output `svg` plots then we use "`set terminal svg`" but if we want plots suitable for inclusion in a LaTeX report then we use "`set terminal epslatex`". The "`set terminal`" command is not hard-coded into each template; instead it is added at runtime. This means that only a single line needs to be changed in order to change every plot from `svg` to LaTeX format.

Output files are saved to the `data/output/` directory. Three files are saved for each plot:

.dat The raw data file.

.gnu The `gnuplot` script generated by instantiating the relevant template file.

.svg / .eps The image file containing the plot.

5.2. Limitations and future work

Using the UNIX text-processing utility `sed` for token replacement ties this implementation to the UNIX platform (because `sed` is not available on some platforms). An improvement would be to use the Boost C++ String Algorithms Library to do the token replacement instead of `sed`.

6. Final design iteration: graphs and spikes

6.1. Introduction

As work progressed on the previous design, it became clear that it would fail to take full advantage of two key features of the disaggregation problem. In the following we present some background before discussing these two features in detail.

As discussed in section 3.4.2, the "least mean squares" approach to matching the washing machine's signature against its fingerprint in the aggregate data fails because unknown devices change state over the course of the washing machine's fingerprint which significantly distorts the fingerprint (see figure 6.1 panels *a-c*). Hence we cannot rely on the *absolute* value of the aggregate signal because the baseline constantly moves underneath us.

Given that we cannot rely on the absolute value, can we rely on the *rate of change*? Figure 6.1 panels *d* and *e* show $\Delta aggregate$ and $\Delta signature$, respectively. An interesting observation of the comparison of $\Delta aggregate$ with $\Delta signature$ is that there are about 10 "spikes" in both derivatives which are very similar in value and which punch cleanly through the noise. (The "spikes" in $\Delta signature$ tend to be a little smaller in magnitude than the spikes in $\Delta aggregate$; we will address this shortly). If you squint then the sequence of spikes in $\Delta signature$ look a little like a door key. The analogy does not work very well visually, but the main idea is that the sequence of "spikes in $\Delta signature$" can be used as a unique identifier for each device; an identifier which is both specific enough to distinguish between devices and robust enough to be detected amongst the noise.

A broad outline of the disaggregation algorithm follows: say we're looking for the washing machine's fingerprint. We first look for a spike in $\Delta aggregate$ with a value of 2000, representing the moment the washing machine first turns its heater on. After finding this "start spike" at time t, we search near *t+5minutes* for a spike with a value of -2000 (the moment the washer turns its heater off). If this is found then look a handful of other salient spikes at their expected times and values. If we find all the salient spikes from the washing machine's signature then we can confidently attribute this fingerprint to the washing machine.

After observing that the spikes in $\Delta aggregate$ are highly similar to the corresponding spikes in $\Delta signature$, it became clear that the system must be re-engineered to take full advantage of these "spikes in the derivative". Given that we're committed to a re-design, let us also re-consider how to store this sequence of spikes in a way which represents the physical behaviour of electrical appliances. The concept of "device power states" mentioned in the previous chapter still appears to be a valid approximation for the physical behaviour of appliances. The hypothesis is that each device has a finite set of "power states". Each power state can be followed by a small number of other power states. For example: the washing machine's heater is always followed by a slow turn of the drum; never by a fast spin. Each device has a finite set of power states and a finite set of permissible power state sequences.

How should "spikes" be represented together with power states? In the previous design, power states were represented as a flat list, as were permissible sequences. That seemed a valid approach until it further data collection demonstrated that the washing machine arbitrarily repeats certain sub-sequences (figure 6.2). Supposedly the washing machine constantly measures the water temperature in the drum and if the temperature drops below a certain threshold then it turns the heater on. The precise timing of the heater's state changes, and indeed the number of times the heater turns on, is dependent on external factors like the air temperature outside the washing machine, the incoming water temperature etc. Our disaggregation system requires a data representation format which can elegantly represent repeating sub-sequences.

During Imperial's Robotics course, we were introduced to robot localisation algorithms which represent a robot's physical environment as a *graph*. Rooms are represented as *vertices* (nodes)

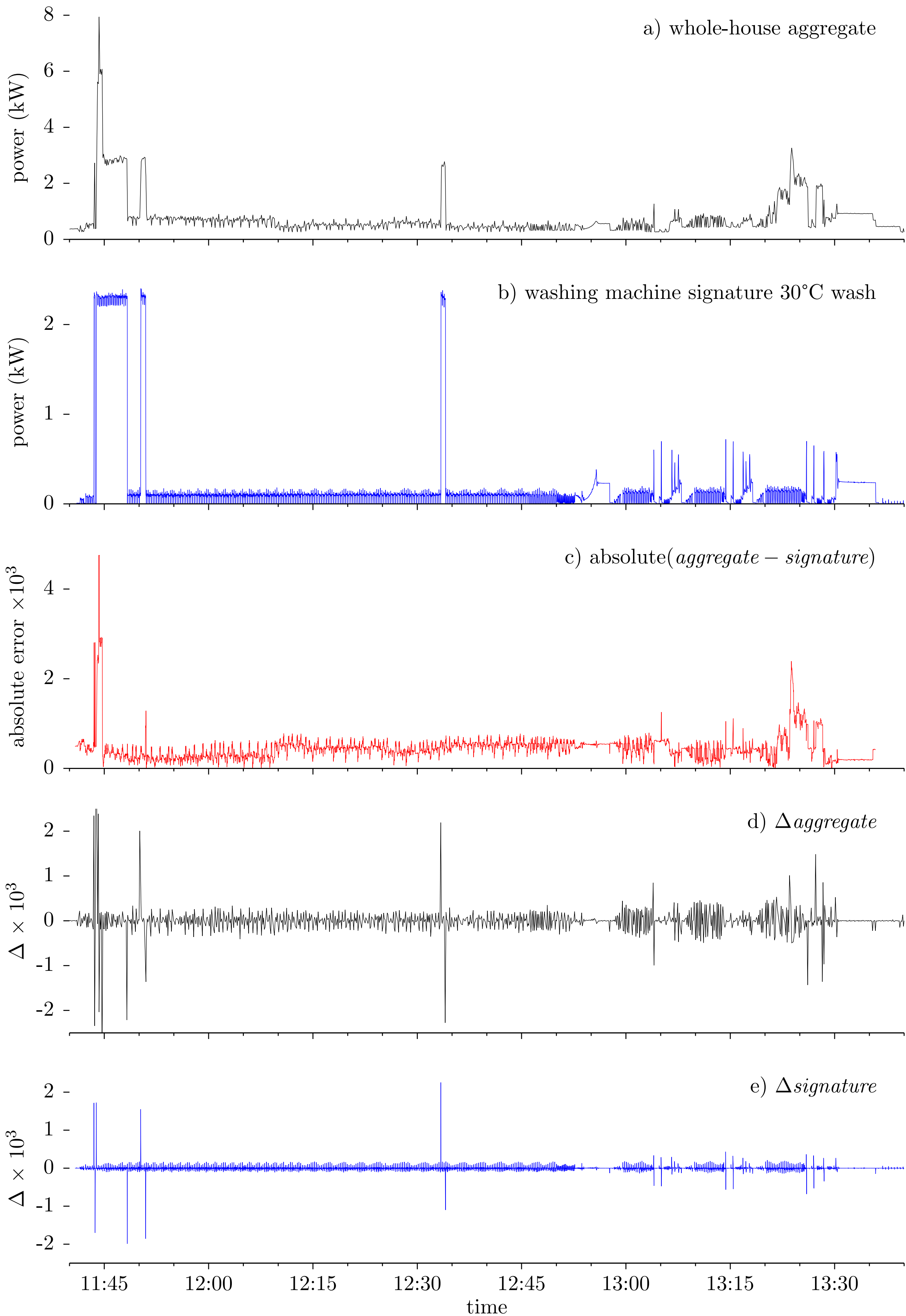


Figure 6.1.: Panels *a*-*c* illustrate why Least Mean Squares fails to locate the washing machine fingerprint (because, as shown in panel *c*, there are large errors around 11:45 and 13:20). (Please note the different y-axis scales in panels *a* and *b*.) Panel *a* shows the aggregate data recorded from the whole house, panel *b* shows the washing machine signature and panel *c* shows absolute error. Panels d & e show the rate of change of the aggregate signal and the rate of change of the signature, respectively.

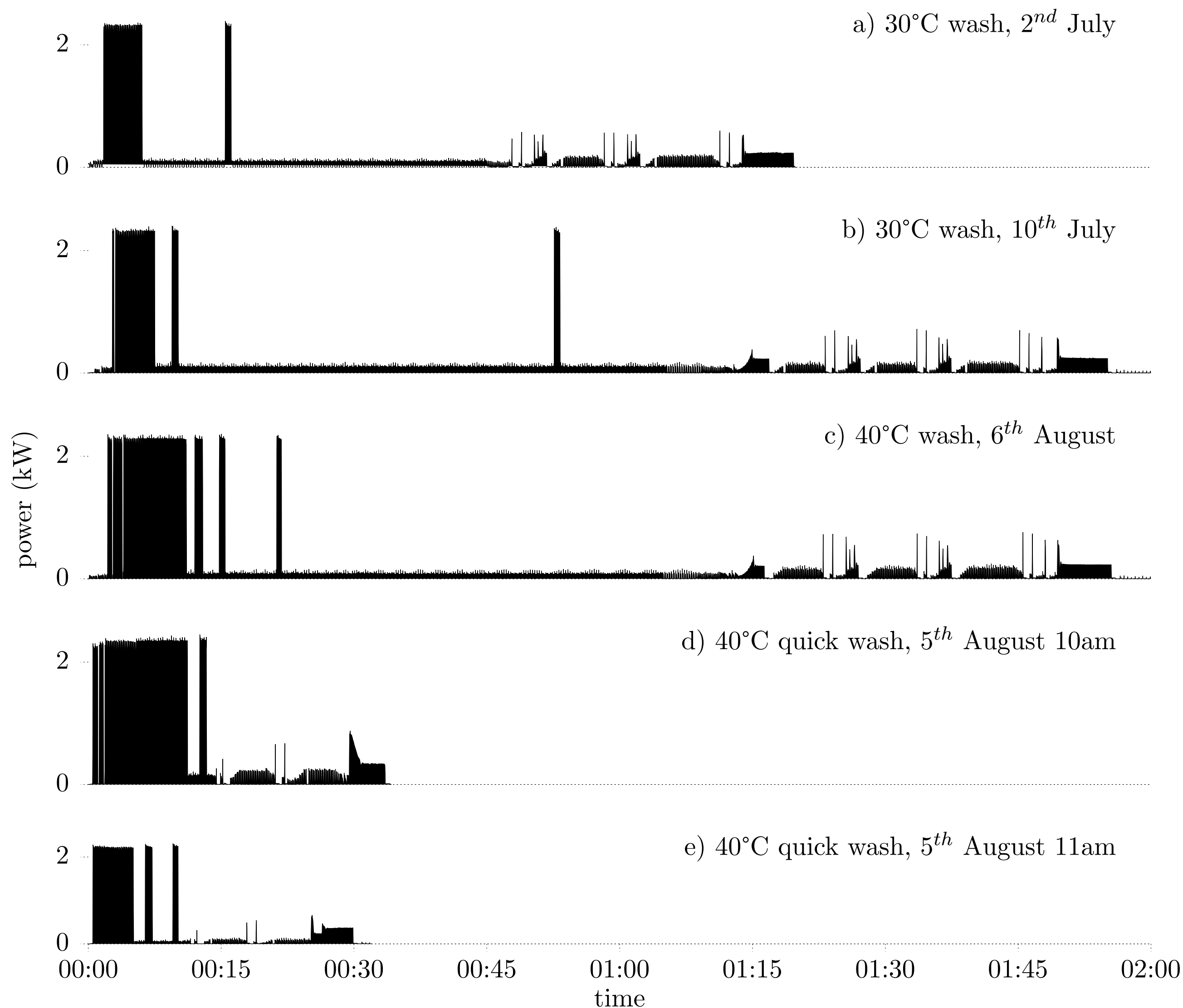


Figure 6.2.: Comparison of 5 different runs of the same washing machine. The 2 kW readings are the water heater. Note the considerable variation between runs.

and permissible routes between rooms are represented as *edges* between vertices. A graph data structure could be used to store device power states as vertices and transitions between these power states as directional edges. The edges of the power state graph perform a similar role to the edges of the robot localisation graph: they represent the permissible "routes" between power states. The permissible routes between power states are the "spikes in Δ*signature*" which lie at the boundary between power states. As such, an edge in our power state graph between vertex v and w must capture information about the spike observed in Δ*signature* during the transition from power state v to power state w. And because there is a considerable amount of noise in the signals, each quantity must be represented statistically by a *mean*, *minimum, maximum* and *standard deviation.* The basic idea is illustrated in figure 6.3 for a kettle.

6.2. Design

6.2.1. Overview of training algorithm

During training, the aim is to create a new power state graph from a set of raw device signatures. The graph vertices represent the power state values. The graph edges are directional, having a *source* and a *target* vertex. The graph edges represent the conditions required to transition from

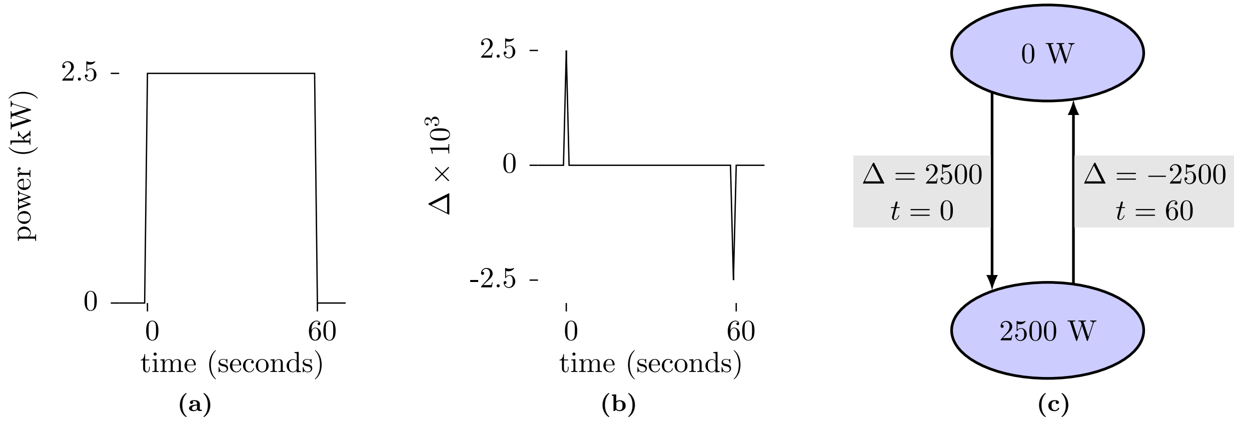


Figure 6.3.: The basic steps for producing a power state graph for a kettle. First take the raw signature (*a*). Next calculate Δ*signature* (*b*). Then produce a power state graph (*c*) where the vertices represent the device power states and the edges represent the conditions necessary to transition between power states.

one power state to another. In particular, the edges represent the expected timing and value of the Δ spikes which indicate a transition between power states.

The first step is to calculate Δ*signature* which is simply done by subtracting $sample_i$ from $sample_{i+1}$. Next select the 10 most "salient and robust" spikes in Δ*signature* (exactly what "salient and robust" means will be discussed below but for now it's sufficient to equate "salient" with "largest magnitude"). We next need to determine the power states which lie either side of each spike. For each $spike \in \Delta signature$, we return to the raw device signature and create a statistical representation for a set of samples immediately before the spike and another statistical representation for a set of samples immediately after the spike. We then search the existing vertices of the power state graph[1] to see if any vertex already represents either the *"pre spike stats*" or the "*post spike stats*". If no matching vertex is found then create a new one. Once a *source* vertex and a *target* vertex has been identified or created, create an edge from *source* to *target* and record the value and timing of *spike* in the edge.

An illustration of the training algorithm in action is show in figure 6.4. As can be seen from the signature in figure 6.4, this device has 4 power states: 0, 100, 200 and 1,000 W, hence there are 4 vertices in the power state graph (panel *b*). This device cycles several times between the 100 W power state and the 1,000 W power state, hence there is a cycle in the power state graph between the 100 W and 1,000 W vertices.

Let us step through part of the signature to gain a clear understanding of how the power state graph is created. The first "spike" in Δ*signature* happens 0 seconds after the start of the signature and has a magnitude of 100 (because it transitions the device from 0 W to 100 W), hence we record "Δ = 100, t=0" on the edge from "0 W" to "100 W". The second spike in Δ*signature* happens 60 seconds after the first spike, has a Δ of 900 and marks the transition from 100 W to 1,000 W. This spike is recorded on the edge from 100 W to 1,000 W. There are a further 2 transitions from 100 W to 1,000 W later in the signature: one at 170 seconds (30 seconds after the preceding spike) and one at 300 seconds (90 seconds after the preceding spike). Recall that each variable is captured not as a single scalar value but as a statistical representation. The edge from 100 W to 1,000 W needs to capture 3 different durations: 30s, 60s and 90s, hence the statistics for the edge from 100 W to 1,000 W are: $t_{\text{min}} = 30$, $t_{\text{mean}} = 60$, $t_{\text{max}} = 90$.

Merge spikes

In section 6.1, we noted that some spikes in Δ*signature* are smaller in magnitude than the equivalent spike in Δ*aggregate*. This anomaly is troublesome given that a core requirement of our

[1] One nice feature of this approach is that exactly the same code is used to both create a new power state graph from scratch and to update an existing power state graph from a new signature.

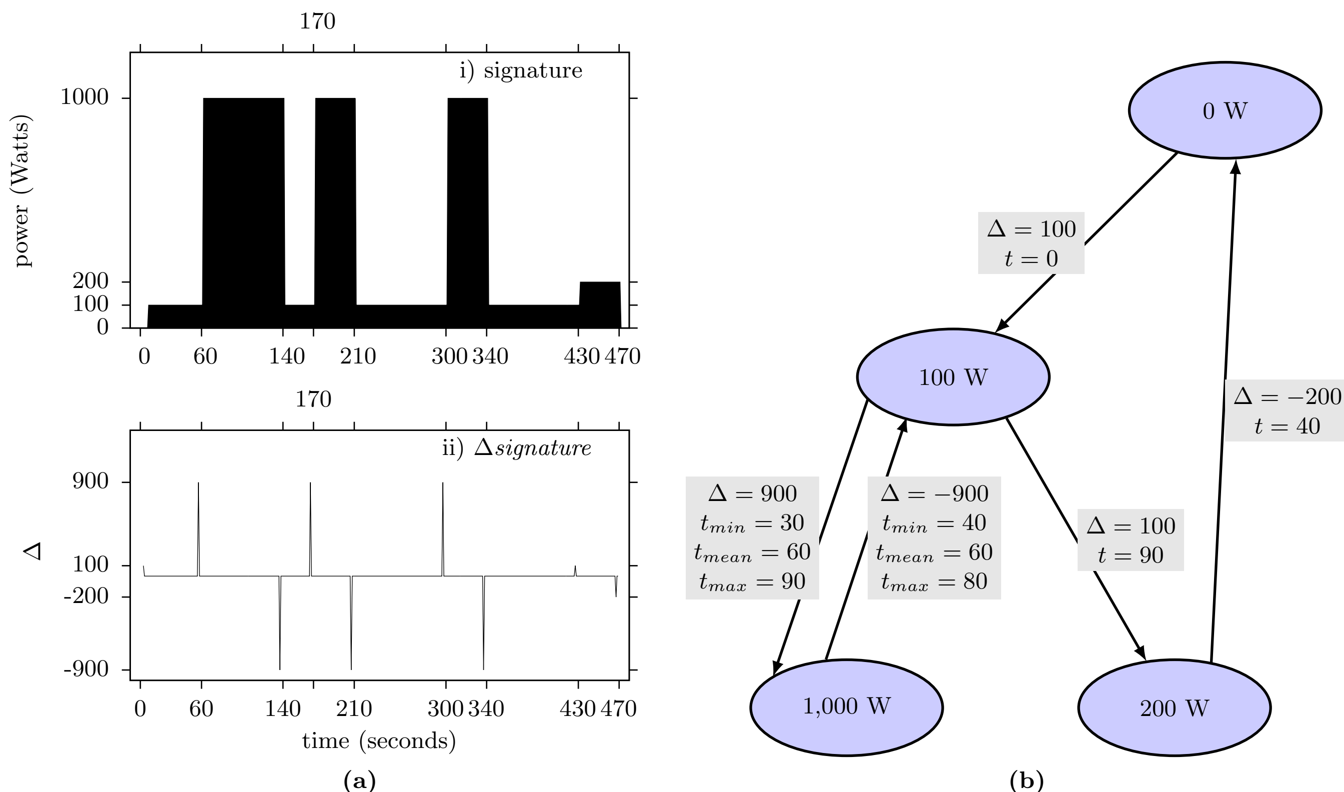


Figure 6.4.: An illustration of the production of a power state graph from a signature which includes repeating power state transitions. Panel *a(i)* shows the raw signature (which has been fabricated to look a little like a washing machine signature). Panel *a(ii)* shows Δ*signature*. Panel *b* shows the power state graph generated from the raw signature.

disaggregation system is that it must be able to reliably locate spikes in Δ*aggregate* of a specific size. Why are some spikes smaller in Δ*signature* than in Δ*aggregate*?

Figure 6.5 shows a detailed comparison of spikes in Δ*aggregate* and Δ*signature*. The spike at 12:33:24 is of almost exactly the same magnitude in both Δ*aggregate* and Δ*signature* but the spike at 12:34:03 is -2000 in Δ*aggregate* and only -1000 in Δ*signature* (i.e. half the magnitude). Recall that the aggregate signal is sampled once every 6 seconds whilst the signature is sampled once a second. For some reason, the device takes 2 seconds to transition between power states at 12:34:03 and so this transition is spread across 2 samples in the aggregate signal.

The solution is to "merge" spikes of the same sign into a single spike. This merging simply searches for consecutive Δ*signature* data points and adds together consecutive data points with the same sign.

Remove fleetingly transient spikes

One of our broad aims is to identify spikes in Δ*signature* which have a good chance of being found in Δ*aggregate*. One problem with the aggregate signal is that, at best, it is sampled at a sixth of the frequency at which the signature is sampled. Worse, the Current Cost smart meter which records the aggregated signal occasionally drops a reading.

The end result is that we need to find power states in *signature* which last longer than, to be safe, 18 seconds. A simple way to do this is to remove from Δ*signature* all equal and opposite spikes within 18 seconds of each other.

We also remove any merged spikes with a magnitude of less than 10 because these are unlikely to be found reliably in the aggregate signal (this does mean that, unfortunately, the current design has no hope of disaggregating devices with a power consumption of 10 Watts or less).

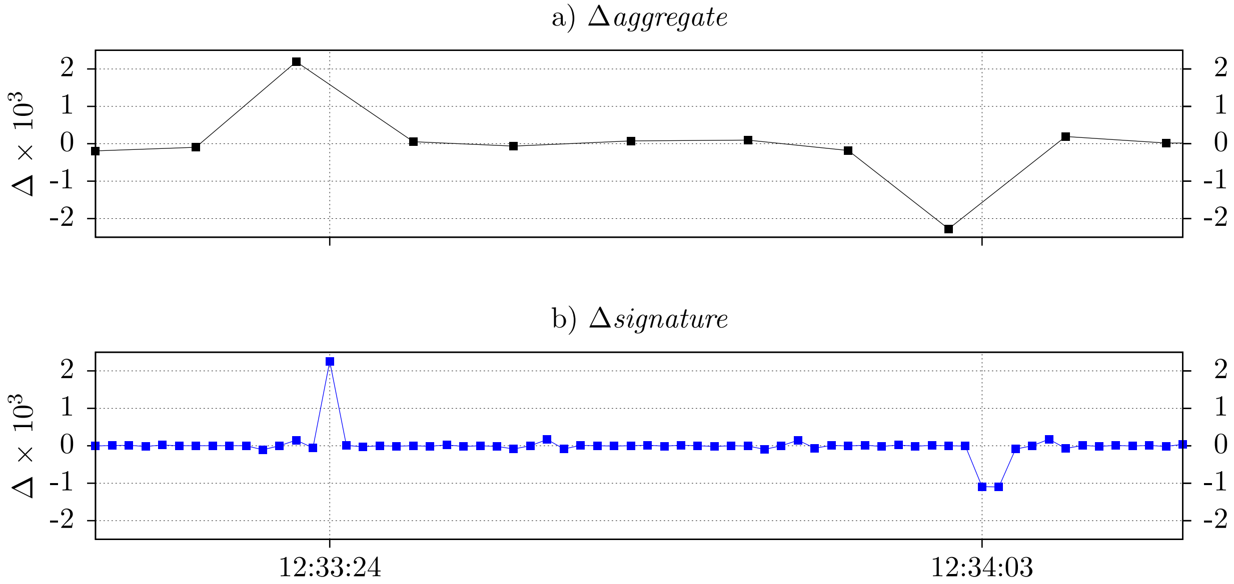


Figure 6.5.: A detailed comparison of "spikes" in $\Delta aggregate$ and $\Delta signature$. The aggregate and signature data were recorded simultaneously.

Reject power state transitions between highly similar power states

The ideal power state transition is one which moves between two power states with radically different means and tiny standard deviations. A small standard deviation is desirable because this indicates that the power state is relatively constant which will ease identification in the noisy aggregate signal. A little bit of experimentation was required to come up with a set of criteria for rejecting spikes. The exact criteria are given in algorithm 6.1, step *6c*.

The full training algorithm is given in algorithm 6.1.

6.2.2. Overview of disaggregation algorithm

Once a complete power state graph has been created, how can this be used to locate the device's fingerprint in an aggregate signal?

Every power state graph has a "0 W" vertex (the *"off"* vertex; vertex 0). This "*off*" vertex has both an *out*-edge (representing the start of the device's run) and at least one *in*-edge (representing the end of the device's run). The aim during disaggregation is to complete a round-trip through the power state graph, starting and finishing at the "off" vertex. The disaggregation algorithm starts by searching $\Delta aggregate$ for the spike encoded by the out-edge from the "off" vertex. Once the algorithm finds this "start spike", it calls the helper algorithm *initTraceToEnd(vertex 1)* which, in turn, calls the recursive algorithm *traceToEnd*.

traceToEnd searches the aggregate signal for all spikes which constitute a permissible *power state transition* to another vertex. This algorithm takes *vertex v* as an input parameter and starts by obtaining a list of every *out-edge* from vertex *v*. For each $edge \in outEdges$, we search $\Delta aggregate$ for the spike encoded by *edge*. If this spike is found then we let $target \leftarrow target_vertex_pointed_to_by_edge$ and call *traceToEnd(target)*. A *likelihood* is calculated for each spike by passing the spike's actual magnitude to the *probability density function* for the spike's statistical representation. (This *likelihood* is normalised by dividing by the *likelihood of the mean*). In a similar fashion, a likelihood is calculated for the *time* the spike was located. The likelihood for magnitude and the likelihood for temporal location are averaged together and recorded.

Every hop completed by *traceToEnd* is recorded in a "disaggregation tree". This is an acyclic graph which records all the possible ways in which the power state graph can be "fitted" to the aggregate data. The vertices of the "disaggregation tree" represent each power state found in the aggregate data. In particular, each vertex records the start timestamp and the Wattage for each power state. Each edge records the *likelihood* that the hop from the *source* to the *target*

Algorithm 6.1 Training algorithm for creating or updating a powerStateGraph from a signature

1. Prepare a list of *spikes*:

   a) Let Δs_i be $sample_{i+1} - sample_i$ for all $sample \in Signature$

   b) Let *mergedSpikes* be a list of "merged" spikes. "Merging" finds consecutive values of Δs with the same sign and adds them together. (See section 6.2.1 in the main text). Each merged spike in *mergedSpikes* is represented as a tuple of the form $(index,\ n,\ merged\Delta s)$ where

   $$merged\Delta s = \sum_{j=i}^{i+n} \Delta s_j$$

   where i is the start *index* and n is the number of consecutive samples "merged".

   c) Erase from *mergedSpikes* all spikes with $\Delta s < 10$

   d) Erase from *mergedSpikes* any fleetingly transient spikes (see section 6.2.1): For each $spike \in mergedSpikes$, let *spikesAhead* be a set of all *spikes* up to 20 seconds forward in time from *spike*. If any $spikeAhead \in spikesAhead$ has a $merged\Delta s$ of opposite sign and a magnitude within 20 % of the magnitude of $spike.merged\Delta s$ then erase both *spike* and *spikeAhead*.

   e) Let *spikes* be the 10 *mergedSpikes* with the greatest magnitude $merged\Delta s$

2. Let *powerStateGraph* be a graph whose vertices store statistics describing a power state and whose edges store the pair *(deltaStats, durationStats)* where *deltaStats* is a statistical representation of a set of $spike.merged\Delta s$ and where *durationStats* is a statistical representation of the length of time this edge's source power state should last.

3. Create a vertex for representing "off", *offVertex*

4. Let $sourceVertex \leftarrow offVertex$

5. Let $indexOfLastAcceptedSpike \leftarrow 0$

6. For each $spike \in spikes$ do:

   a) Let *preSpikePowerState* be a statistical representation of the 8 raw samples before *spike.index*

   b) Let *postSpikePowerState* be a statistical representation of the 8 raw samples after $spike.index + spike.n$

   c) Reject this *spike* and continue to the next *spike* if any of the following criteria are met:

      i. $preSpikePowerState.stdev > preSpikePowerState.mean$

      ii. $postSpikePowerState.stdev > postSpikePowerState.mean$

      iii. *preSpikePowerState.mean* and *postSpikePowerState.mean* are within $0.2 \times \max(preSpikePowerState.mean,\ postSpikePowerState.mean)$

   d) Add or update a vertex. Search through each $vertex \in powerStateGraph$ looking for a *vertex* which has a similar mean (assessed by T-Test) to *postSpikePowerState*. If a similar existing *vertex* is found then update that *vertex* with the raw data used to make *postSpikePowerState* and set $targetVertex \leftarrow vertex$. If no existing *vertex* is found then create a new *vertex* with values taken from *postSpikePowerState* and set $targetVertex \leftarrow vertex$.

   e) Add or update an edge. If an edge already exists from *sourceVertex* to *targetVertex* then update its stats, else add a new one with $deltaStats \leftarrow spike.merged\Delta s$ and $durationStats \leftarrow spike.index - indexOfLastSpike$

   f) $indexOfLastSpike \leftarrow spike.index$ and $sourceVertex \leftarrow targetVertex$

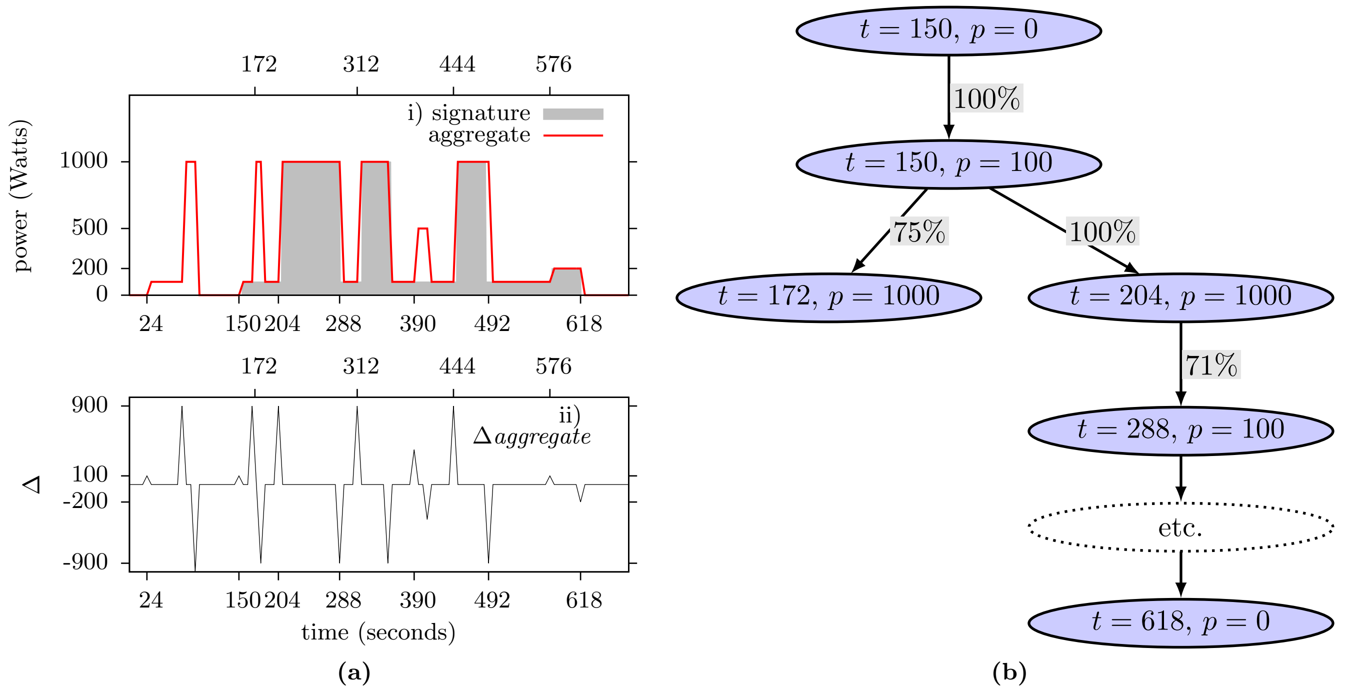


Figure 6.6.: Locating our synthetic washing machine signature in a synthetic aggregate signal. Panel *a(i)* shows the aggregate signal in red and the signature we're trying to locate in solid grey. Panel *a(ii)* shows $\Delta aggregate$. The system first produces the power state graph shown in figure 6.4 from the device signature and then uses this power state graph to locate this device's fingerprint in the aggregate signal. The tree structure in panel *b* shows the "disaggregation tree" produced when the system attempts to determine if $t = 150$ is the beginning of a complete fingerprint of the device (which it is). The vertices represent power states located in the aggregate signal. The two variables stated in the graph vertices are t (start time) and p (power in Watts). The values associated with each edge are average likelihoods that the spike found in $\Delta aggregate$ is the correct value and time. The vertices between $t = 288$ and $t = 618$ have been omitted to save space. (In panel *a(i)*, the signature does not perfectly align with the aggregate signal because the aggregate signal has a sample period of 6 seconds, whilst the signature has a sample period of 1 second.)

vertex is correct. Once the disaggregation tree is completed, a simple "recursive flood" algorithm runs through every complete path in the disaggregation tree finding the path with the highest mean likelihood. This path is then reported to the user, along with the path's duration and the estimated energy consumption of the device. The broad disaggregation approach is illustrated in figure 6.6.

Let us briefly run through the disaggregation illustrated in figure 6.6. The first step is to obtain the spike value encoded by the out-edge from the "off" vertex in the power state graph. In the case of this illustration, a value for Δ of 100 marks the start of a device run. The disaggregation algorithm searches $\Delta aggregate$ for times where $\Delta aggregate = 100$. The three times where $\Delta aggregate = 100$ are $t = 24$, 150 and 576. The system then "follows through" on each of these start times, using the function *traceToEnd* to attempt to complete a round-trip through the power state graph back to "off". The disaggregation tree produced by "following through" from the start spike at $t = 150$ is given in figure 6.6 panel *b*. The algorithm has already identified that the spike at $t = 150$ may represent a transition from "off" to "100 W", hence there is a vertex marked "$p = 100$" immediately down-stream of the off vertex. The spike is precisely the correct magnitude, hence the likelihood marked on the edge from the off vertex is "100 %". Where do we go from this power state?

The power state graph dictates that there are two possible ways to transition away from the "100 W" vertex: either a spike with magnitude 900 at 30-90 seconds ahead, or a spike with magnitude 100 at 90 seconds ahead. *traceToEnd* searches for spikes which fit these parameters. Spikes with a value of 900 within the expected time window are found at $t = 172$ and $t = 204$ (hence the branch in the disaggregation tree). These are candidates for a transition from the "100 W" power

Algorithm 6.2 Main algorithm for disaggregating `aggregate` data using `powerStateGraph`

1. Let *startSpike* be the spike described by the first edge of *powerStateGraph*
2. For every *possibleStartSpike* in Δ*aggregate* matching *startSpike:*
   a) Let *candidateFingerprint* ← *initTraceToEnd(possibleStartSpike)*
   b) If *initTraceToEnd* succeeds then add *candidateFingerprint* to *fingerprintList*
3. Remove overlapping fingerprints. Fingerprints A and B overlap if $A.startTime < B.startTime < A.endTime$. If A and B overlap then remove the *fingerprint* with the lowest confidence. Repeat until no *fingerprints* overlap. (The assumption is that each household only has a single instance of each device and hence overlapping device signatures must not be allowed. If multiple instances of the same device do indeed exist then overlapping fingerprints can be kept by supplying `disaggregate` with a `--keep-overlapping` argument.)
4. Return *fingerprintList*

state to "1,000 W". *traceToEnd* calls itself twice: once for $t = 172$ and once for $t = 204$. Both of these subsequent instantiations of *traceToEnd* search for a spike where Δ*aggregate* $= -900$ within the specified time window of 40-80 seconds. The instance of *traceToEnd* attempting to "follow through" on $t = 172$ fails to find a suitable spike 40-80 seconds ahead of $t = 172$ and so gives up. But the instance of *traceToEnd* attempting to "follow through" on $t = 204$ successfully finds a -900 spike at $t = 288$ and calls *traceToEnd* again. After several more recursions, the "off vertex" is successfully located at $t = 618$. In this simple example, only one complete path exists in the disaggregation tree and so this path is reported to the user.

Estimating energy consumption & recording inter-spike power statistics

One nice feature of the "graph and spikes" approach is that it allows us to estimate the power consumption of each fingerprint with very little effort because the disaggregation algorithm already collects all the information we need: the mean power consumption and the duration of each power state. Estimating total energy consumption for each power state in the disaggregation tree is simply a case of multiplying the power consumption (1 Watt is 1 Joule per second) by the power state duration (in seconds) to arrive at a total energy consumption in Joules. The most common unit for describing domestic energy consumption is the kilowatt hour (kWh). An energy consumption figure in Joules can be converted to kilowatt hours (kWh) by dividing by 3.6×10^6.

Recall that the training algorithm defines each power state by creating a statistical representation of only 8 samples immediately preceding and following each spike. Testing demonstrates that this works well for training the power state graph and even provides a respectable basis for estimating the total energy consumed by each device run. However, the power consumption of complex devices like washing machines fluctuates considerably between each spike (recall that we only take 10 spikes from each signature yet a washing machine signature may be up to 2 hours in length; we cannot assume that the washing machine's power consumption remains constant between these 10 spikes). To do a better job of estimating the power consumed by each device run, we also store statistics for every signature sample *between* adjacent spikes. These "betweenSpikes" statistics have a huge standard deviation and hence are not suitable for use as the primary definition of a "power state". The "betweenSpikes" statistics are *only* used for estimating energy consumption.

The top-level disaggregation algorithm is given in algorithm 6.2, the algorithm for *initTraceToEnd()* is given in algorithm 6.3 and the algorithm for *traceToEnd()* is given in algorithm 6.4.

Algorithm 6.3 Algorithm for *initTraceToEnd(spike)*

1. Create a new *disaggregationTree.* A path through *disaggregationTree* from the root vertex to a leaf vertex represents one possible "fit" of the *powerStateGraph* to the *aggregateData*, starting at a single *possibleStartSpike.* The root vertex of the *disaggregationTree* always represents "off". Tree vertices hold a *timestamp* and a pointer to a *powerStateGraph* vertex. Tree edges hold a *likelihood* represented as a float.

2. Add *offVertex*

   a) Let *offVertex.timestamp* ← *possStartTimeOfFingerprint* where:

      possStartTimeOfFingerprint ← *spike.timestamp* − *powerStateGraph[edge0].duration*

   b) Let *offVertex.psgVertex* ← *powerStateGraph[vertex0]*

3. Add *firstVertex*

   a) Let *firstVertex.timestamp* ←*spike.timestamp*

   b) Let *firstVertex.psgVertex* ←*powerStateGraph[vertex1]*

4. Add *edge* from *offVertex* to *firstVertex.* Let *edge.likelihood* ← *spike.likelihood*

5. *traceToEnd(disaggregationTree, firstVertex).* This populates *disaggregationTree* by recursively finding every possible fit of *powerStateGraph* to *aggregateData*

6. Let *listOfPaths* be a flat list of all complete paths from *disaggregationTree[offVertex]* to another *offVertex.* This flat list is produced by the "recursive flood" algorithm *findListOfPathsThroughDisagTree*

7. Return the path in *listOfPaths* with the highest average likelihood

Algorithm 6.4 Recursive algorithm for *traceToEnd(disaggregationTree, disagVertex)*

1. Base case: *if (disaggregationTree[disagVertex] == offVertex)* then return.
2. Let *psgOutEdges* be a list of all out edges from the *powerStateGraph* vertex pointed to by *disagVertex.psgVertex*
3. For each $psgOutEdge \in psgOutEdges$ search for all spikes in $\Delta aggregate$ where $\Delta aggregate = psgOutEdge.\Delta s$ within a specific search window. Specifically, do this:

   Calculate the boundaries of the search window:

   a) Let $e \leftarrow$ *powerStateGraph[psgOutEdge].duration.stdev*
   b) Let *beginningOfSearchWindow* $\leftarrow$ *disaggregationTree[disagVertex].timestamp* + *powerStateGraph[psgOutEdge].duration.min* $- e$
   c) Let *endOfSearchWindow* $\leftarrow$ *disaggregationTree[disagVertex].timestamp* + *powerStateGraph[psgOutEdge].duration.max* $+ e$
   d) Let list *foundSpikes* be all spikes in $\Delta aggregate$ where $\Delta aggregate =$ *powerStateGraph[psgOutEdge].*Δs between *beginningOfSearchWindow* and *endOfSearchWindow*
   e) For each $spike \in foundSpikes$ create a new vertex in *disaggregationTree* and recursively *traceToEnd* this new vertex. Specifically, do this:

      i. Calculate a normalised *likelihoodForTime* of finding *spike* at *spike.timestamp*

      ii. Let *averageLikelihood* $\leftarrow$ *(likelihoodForTime + spike.likelihood)* $\div$ *2*

      iii. Add *newVertex* to *disaggregationTree*

      - Let *newVertex.timestamp* $\leftarrow$*spike.timestamp*
      - Let *newVertex.psgVertex* $\leftarrow$*target(psgOutEdge)*

      iv. Add a *newEdge* from *disagVertex* to *newVertex*

      - Let *newEdge.likelihood* $\leftarrow$*averageLikelihood*

      v. *traceToEnd(disaggregationTree, newVertex)*

6.3. Implementation

A quick reminder that the Doxygen documentation for the code is available at
www.doc.ic.ac.uk/∼dk3810/disaggregate

6.3.1. Maintaining "legacy" functions and classes

The "Graphs and Spikes" design iteration represents a significant shift away from the previous design so a new `git` branch was started. Should the irrelevant classes and methods be completely removed from this branch so the new design can be free from any "legacy" code?

The existing `Array` and `Signature` classes and the `GNUplot` and `Utils` helper code can be exploited for the new design with little modification. The `Statistic, Array, Device` and `AggregateData` classes are useful but will need to be given several important new methods (as will be discussed below).

The `Histogram` and `PowerStateSequence` classes are simply no longer relevant. My gut instinct was to remove these completely but I did not do this for 2 reasons: firstly, I wanted to refer to these functions in the report chapters discussing previous design iterations. Secondly, I wanted to keep the previous designs functional so the user could specify at the command line which "mode" to use (the default mode is the "graph and spikes" mode. The command-line switches `--lms` and `--histogram` switch to the relevant modes (see the User Guide in appendix A)).

A new `PowerStateGraph` class was created to do the bulk of the new "graphs and spikes" work. `PowerStateGraph` is responsible for creating a new power state graph from a `Signature` and for disaggregating an `AggregateData` signal.

A UML diagram showing all the functions relevant to the "graphs and spikes" design iteration is shown in figure 6.7.

6.3.2. Boost graph library

Both the "power state graph" and the "disaggregation tree" are implemented using the Boost Graph Library[2]. This is a remarkably powerful library.

The Boost Graph Library allows users to specify custom classes for graph vertices and edges (these are called "bundled properties").

First we define a `struct` for storing power state statistics in each graph *vertex*. Recall that we store two sets of statistics for each power state: the `postSpike` stats for the 8 samples immediately after each spike (used by the main training algorithm) and `betweenSpikes` stats for the sole purpose of estimating energy consumption. Here is the definition of our custom `PowerStateVertex` structure:

```
struct PowerStateVertex {
    /* Stats for 8 samples immediately after the spike.
     * Used for determining vertices during training. */
    Statistic<Sample_t> postSpike;

    /* Stats for the entire period between spikes.
     * Used only for estimating energy consumption.   */
    Statistic<Sample_t> betweenSpikes;

    /* (Sample_t is simply a typedef for a double.)   */
};
```

Next we define a `struct` for the power state graph *edge*. Each edge stores two sets of statistics: the Δ*signature* value and the duration of the power state defined by the *source* edge. In other words, the edge represents the conditions necessary for leaving the power state represented by the edge's source vertex.

```
struct PowerStateEdge {
   Statistic<double> delta;
   Statistic<size_t> duration;
};
```

Finally, we create a graph type which uses our two custom classes:

```
typedef boost::adjacency_list<
        boost::vecS, boost::vecS, // multiple edges between vertices
        boost::directedS,         // we need directional edges
        PowerStateVertex,         // our custom vertex (node) type
        PowerStateEdge            // our custom edge type
> PSGraph;
```

`boost::vecS` tells Boost Graph Library to use a `std::vector<>` as the container for representing each *edge-list* associated with each vertex; the end result is that `vecS` allows each vertex to have multiple in- and multiple out-edges, which is what we want. However, `vecS` also allows edges to exist which do two things we don't want (figure 6.8): 1) edges which leave and enter a single vertex and 2) two vertices can have multiple edges in the same direction between them. These two problems are solved by running two checks prior to adding a new edge. We only add a new edge

[2] boost.org/doc/libs/1_42_0/libs/graph/doc/index.html

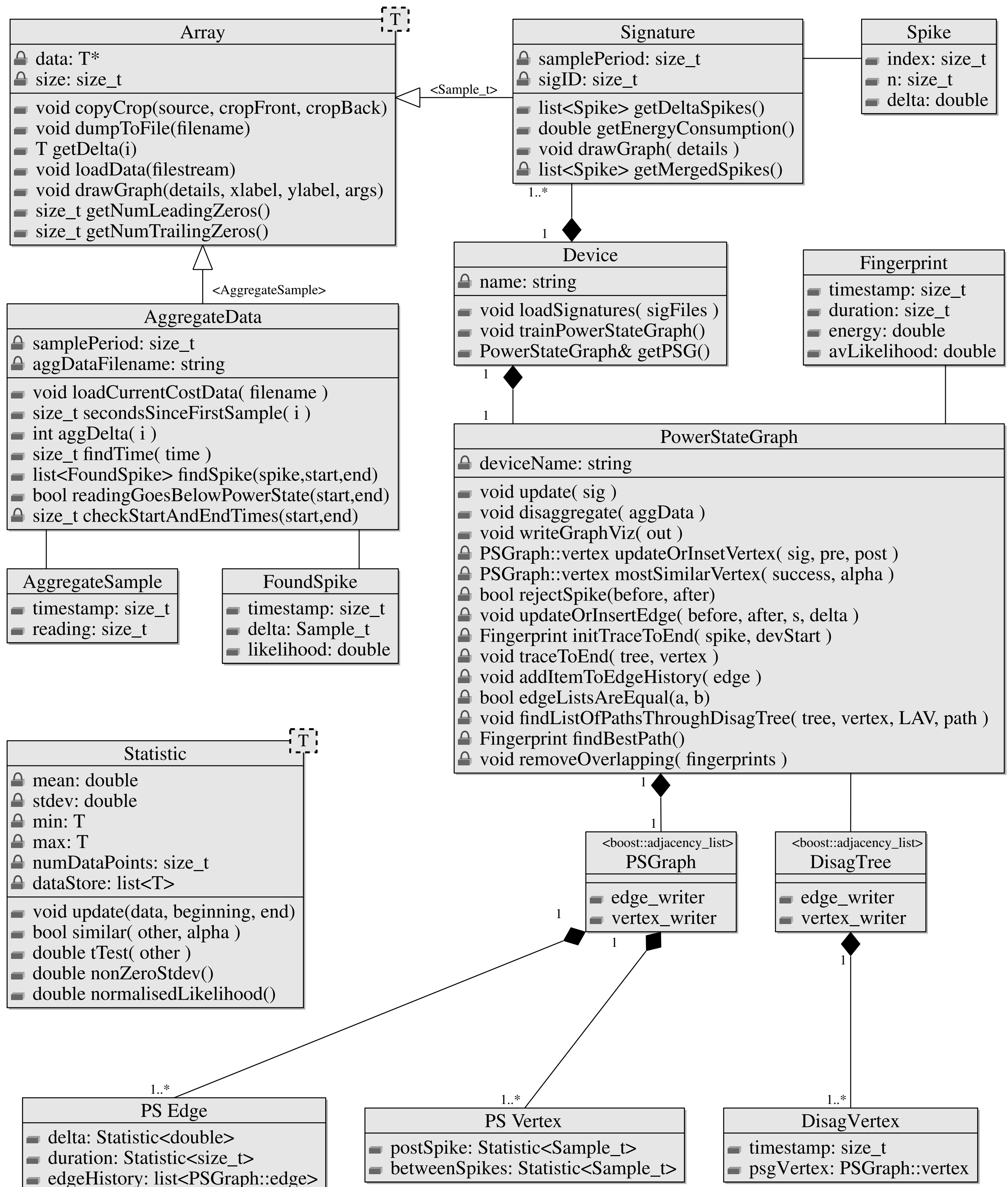
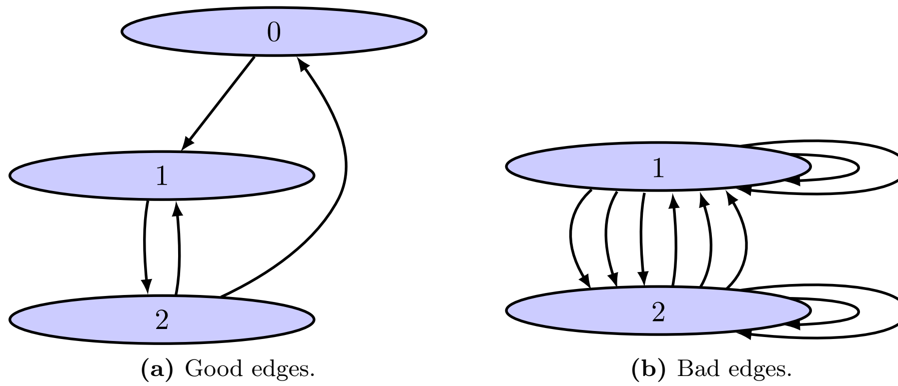


Figure 6.7.: UML Diagram for “Graphs and Spikes” design iteration. `Sample_t` is a type for representing samples and is just a `typedef` for a `double`.



(a) Good edges. **(b)** Bad edges.

Figure 6.8.

if the new edge's `sourceVertex!=targetVertex`. If an edge already exists from `sourceVertex` to `targetVertex` then we update that edge rather than adding a new edge.

6.3.3. Overview of the `PowerStateGraph` class

(Doxygen HTML documentation for `PowerStateGraph`, including *call* and *caller* graphs is available at www.doc.ic.ac.uk/~dk3810/disaggregate/class_power_state_graph.html)

The main responsibility of the `PowerStateGraph` class is to wrap around a `powerStateGraph` object of type `PSGraph` (defined in the code snippet above). `PowerStateGraph` has 2 broad categories of method: methods for *training* the `powerStateGraph` and methods for *disaggregating* an `AggregateData` signal using the `powerStateGraph`.

Training

`PowerStateGraph::update(const Signature& sig)` is the public interface to `PowerStateGraph's` training algorithm. `PowerStateGraph::update()` first acquires a `list` of delta spikes from `sig.getDeltaSpikes()` and selects the 10 largest spikes. For each `spike`, create stats for `preSpikePowerState` and `postSpikePowerState`. Next, check to make sure the `pre-` and `post-SpikePowerState` stats have means which are sufficient far apart. If so then update or insert a new vertex to represent the `postSpikePowerState`. Then update or insert a new edge to represent the `spike`. The simplified code is below (for the purposes of this simplified code, the code for maintaining the `betweenSpikes` statistics has been omitted):

```
void PowerStateGraph::update(const Signature& sig)
{
  list<Signature::Spike> spikes = sig.getDeltaSpikes();
  sourceVertex = offVertex;
  indexOfLastAcceptedSpike = 0;

  // for each spike, locate 8 samples immediately before and after the spike
  for (spike = spikes.begin(); spike!=spikes.end(); spike++) {

    // calculate the start index for the 8 pre-spike samples
    size_t start = ((spike->index > 8) ? (spike->index - 8) : 0 );

    // calculate the end index for the 8 post-spike samples
    size_t end = spike->index + 8 + spike->n + 1;

    // Create statistics for pre- and post-spike
    Statistic<Sample_t> preSpikePowerState(sig, start, spike->index);
    Statistic<Sample_t> postSpikePowerState(sig,
```

```
                                    (spike->index + spike->n + 1), end);

      if (!rejectSpike(preSpikePowerState, postSpikePowerState)) {
        targetVertex = updateOrInsertVertex(sig, postSpikePowerState);

        if (sourceVertex != targetVertex)
          updateOrInsertEdge(sourceVertex, targetVertex,
                (spike->index - indexOfLastAcceptedSpike), spike->delta);

        sourceVertex = targetVertex;
        indexOfLastAcceptedSpike = spike->index;
      }
    }
}
```

Simplified code listings for `updateOrInsertVertex()`, `updateOrInsertEdge()` and `mostSimilarVertex()` are available in appendix B.1.

Disaggregation:

`PowerStateGraph::disaggregate(const AggregateData& aggregateData)` is the public method for disaggregation. This starts by loading the first edge from `powerStateGraph()` which gives us the statistics for the delta spike indicating the start of a device run. Next use `aggregateData.findSpike()` to search for the start delta in $\Delta aggregate$. `findSpike()` returns a list of candidate start spikes. For each candidate start spike, recursively attempt to complete a round-trip through `powerStateGraph`. If this fails then `avLikelihood` is set to `-1`. If we successfully follow through then add this candidate to the list of candidates we return. Finally, if any candidates overlap then remove all overlapping candidates except the candidate with the highest confidence. The simplified code is below:

```
const list<Fingerprint> PowerStateGraph::disaggregate(
                        const AggregateData& aggregateData   )
{

  list<Fingerprint> fingerprintList; // what we return
  Fingerprint candidateFingerprint;
  /* Fingerprint is a struct for bundling start
   * timestamp, duration, energy and avLikelihood */

  /* Load the stats from the first PowerStateGraph edge.
   * The Boost Graph Lib provides an out_edges() function which returns
   * a pair of edge iterators: the first iterator points to the first edge;
   * the second iterator points 1 past the last edge. Dereferencing an
   * edge iterator returns an edge descriptor. tie() allows easy
   * access to each element of the pair of edge iterators. */
  PSG_out_edge_iter out_i, out_end;
  tie(out_i, out_end) = out_edges(offVertex, powerStateGraph);
  PowerStateEdge firstEdgeStats = powerStateGraph[*out_i];

  // Search through aggregateData for possible start spikes
  list<AggregateData::FoundSpike> posStartSpikes =
                aggregateData.findSpike(firstEdgeStats.delta);

  // For each possible start spike in the delta aggregate, attempt
  // to find all the subsequent spikes specified by powerStateGraph edges.
```

```
  for (posStartSpike=posStartSpikes.begin();
       posStartSpike!=posStartSpikes.end(); posStartSpike++) {

    // Attempt to recursively "follow through" from this posStartSpike.
    // initTraceToEnd() calls the recursive function traceToEnd().
    candidateFingerprint = initTraceToEnd( *posStartSpike );

    // If this start spike was successfully traced all the way to
    // an off power state then add this item to fingerprintList.
    if ( candidateFingerprint.avLikelihood != -1 )
        fingerprintList.push_back( candidateFingerprint );
  }

  if ( fingerprintList.empty() ) {
    cout << "No signatures found." << endl;
  } else {
    removeOverlapping( &fingerprintList );
    displayAndPlotfingerprintList( fingerprintList );
  }

  return fingerprintList;
}
```

Simplified code listings for `initTraceToEnd()`, `traceToEnd()` and `findListOfPathsThroughDisagTree()` are available in appendix B.2.

6.3.4. Updating statistics

The "Graph and Spikes" design requires that the `Statistic` class be able to update existing statistics with new data. This is trivial for `mean`, `min` and `max` but is not trivial for `standard deviation`. An early version of the `Statistic::update()` function attempted to get by simply by storing an intermediate value used during the standard deviation calculation. This proved not sufficiently accurate and so each `Statistic` object now stores a copy of every value it has ever been asked to represent. This is a memory-hungry approach and there are likely to be ways to make this more efficient.

6.3.5. Non-zero standard deviation & calculating likelihood

The "average likelihood" reported for each candidate fingerprint is an average of three different likelihoods:

1. The average likelihood of every `spike.timestamp` (i.e. the likelihood of finding a spike with the timestamp the spike was found with)

2. The average likelihood of every `spike.delta` (i.e. the likelihood of finding a spike with the $\Delta aggregate$ value the spike was found with)

3. The likelihood of the entire fingerprint's energy consumption

Each likelihood is calculated using the `boost::math::pdf()` function. The likelihood is then normalised by dividing the raw likelihood by the likelihood of the distribution's mean. The end result is that we get a number between 0 and 1 which tells us how close we are to the distribution's mean; if we're right on target then the normalised likelihood will be 1. To illustrate, here is the code for calculating the normalised likelihood:

```
const double Statistic::normalisedLikelihood( const double x ) const
{
  // Create a normal distribution
  boost::math::normal dist(mean, nonZeroStdev());

  // Calculate normalised likelihood
  return  boost::math::pdf(dist, x) /
          boost::math::pdf(dist, mean);
}
```

What does "`nonZeroStDev()`" do? Given my small training data set, several "statistics" are formed from only a single value and hence have a standard deviation of 0. It makes no sense to try to calculate a likelihood from a "distribution" with a standard deviation of 0. So, as an approximation, `Statistic::nonZeroStdev()` does this:

```
const double Statistic::nonZeroStdev() const
{
  if (stdev < 1 )
    return fabs(mean/10);
  else
    return stdev;
}
```

6.3.6. Indexing aggregate data by timecode

Recall that the Current Cost home energy monitor fails to record approximately 10 % of the samples measured by the sensor; hence we must compensate for the fact that two adjacent recorded samples may not have been sampled precisely 6 seconds apart. To handle this problem as robustly as possible, we index the aggregate data by timecode rather than by array index.

6.3.7. Data output

There are three forms of data output from `PowerStateGraph`. For the purposes of illustration, we'll use the example of disaggregating a washing machine from a single day's aggregate data. Two washing machine signatures will be used to train the power state graph.

Text output to the standard terminal. For example:

```
***** TRAINING POWER STATE GRAPH... *****
Energy consumption from sig0 = 0.307 kWh
Energy consumption from sig1 = 0.403 kWh
Mean energy consumption      = 0.355 kWh

Power State Graph vertices:
0 = {min =    0.0, mean =    0.0, max =  0.0, stdev =  0.0, n =  0}
    (offVertex)
1 = {min = 2239.3, mean = 2324.9, max=2396.8, stdev = 32.1, n = 40}
2 = {min =    2.5, mean =  127.5, max= 282.7, stdev = 62.2, n = 48}
3 = {min =   14.2, mean =   14.3, max=  14.7, stdev =  0.2, n = 32}
4 = {min =  468.7, mean =  517.1, max= 553.7, stdev = 33.2, n =  8}

***** TRAINING FINISHED. DISAGGREGATION STARTING. *****
Finding start deltas... found 47 possible start deltas. Following through...
... done following through.
No candidate fingerprints overlap.
```

```
Candidate fingerprint found:
   timestamp      = 1310294456
   date           = Sun Jul 10 11:40:56 2011
   av likelihood  = 0.89
   duration       = 6902 seconds (1h 55m 2s)
   energy         = 1.37719e+06 Joules equivalent to 0.383 kWh
```

The powerStateGraph is output to `data/output` in two formats: The `graphviz` file describing the `powerStateGraph` is output to `powerStateGraph.gv` and the PDF file produced from the `graphviz` file is output to `powerStateGraph.pdf`. The `graphviz` file is produced using `boost::write_graphviz()` function called with two custom classes for formatting the edges and vertices: `PowerStateGraph::PSG_edge_writer` and `PowerStateGraph::PSG_vertex_writer`. The PDF file is produced by issuing a `system()` call to run the `dot` utility for converting `graphviz` files into graphical files.

GNUplot files: Recall that my GNUplot code produces three files per GNUplot graph: a `.dat` file storing the raw data, a `.gnu` file storing the instantiated GNUplot template and a `.svg` graph. The first two graphs listed use the `1line.template.gnu` GNUplot template.

<signature name>-afterCropping.[svg|dat|gnu] The raw signature file after it has had zeros cropped from the tail and head.

<signature_name>-delta.[svg|dat|gnu] The Δ*signature* data.

disagg.[svg|dat|gnu] This is the end-goal. This shows the aggregate data with each candidate device fingerprint. An example is given in figure 6.9. The GNUplot template used is `disagg.template.gnu`. The x-axis of the graph is scaled so we can see every fingerprint plus a border either side.

6.3.8. Boost program options

The command-line arguments and the `config/disaggregate.conf` file are parsed by the Boost Program Options library. All the code declaring and parsing the options is located in `Main.cpp`.

6.3.9. Refinements

Edge histories

When the code was first run as specified above, it did a good job of disaggregating kettles and toasters but got stuck in hugely lengthy loops (taking many minutes to run) when disaggregating the washing machine. After some time trawling through the log files, it became clear that the `powerStateGraph` was too weakly constrained so the poor disaggregation algorithm found an enormous number of ways to fit the `powerStateGraph` to the aggregate data.

To better constrain the `powerStateGraph`, I added a “rolling edge history” to each edge (see figure 6.10). The disaggregation algorithm is only allowed to travel across `powerStateEdge` edge e if, at that point in time, the disaggregation algorithm has traversed the same sequence of edges as specified in *e.edgeHistory*. For example, the power state graph in figure 6.10 (with an edge history length of 2) constrains the disaggregation algorithm to a single round-trip from vertex 1 to vertex 2 and back to 1. This is because edge `(1,2)` has an edge history of `(0,1)` which means that edge `(1,2)` can only be traversed if the disaggregation algorithm's edge history is also `(0,1)` when it attempts to traverse edge `(1,2)`. Once the disaggregation algorithm has travelled through vertices 0, 1, 2, 1 then its “edge history” will be `(1,2), (2,1)` and hence will not be allowed to traverse the `(1,2)` edge again.

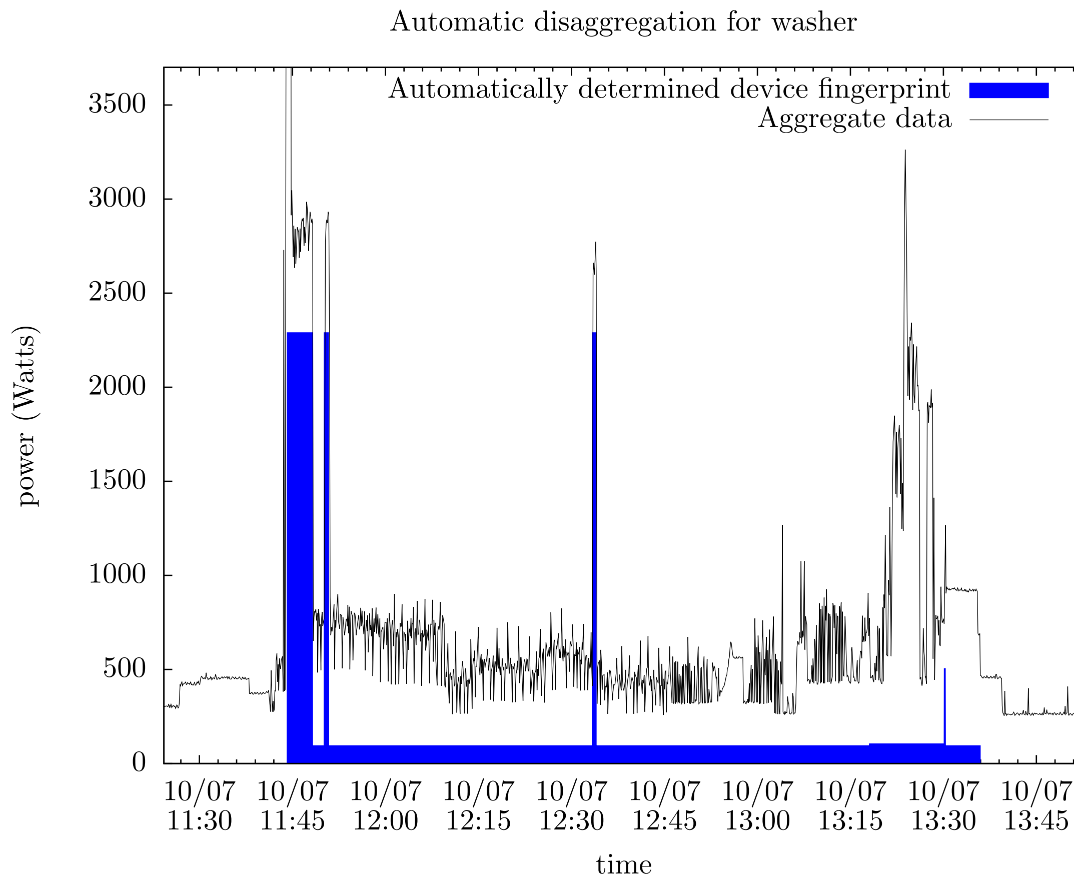


Figure 6.9.: An example of the `disagg` output for a washing machine. This shows the end-goal of the entire system. The system has correctly identified the time and duration of the washing machine fingerprint and has identified the timing for each power state within a single device run.

Ensure the absolute aggregate data value does not drop below power state minimum

Figure 6.11 shows the disaggregation algorithm attempting to fit a powerStateGraph trained on washing machine signatures to a section of aggregate data dominated by the tumble drier. This is a particularly bad answer because the aggregate data signal actually dips *below* the hypothesised device power state at some points. One of our mantras so far has been "the absolute value of the aggregate data signal is not useful". However, the disaggregation algorithm is clearly failing when it estimates that a device fingerprint exists where the aggregate signal drops *below* the signal required by the fingerprint. The aggregate signal is a sum of all devices and so *cannot be less than* the power consumption for a single device!

To fix this faulty logic, the *traceToEnd()* function was modified to check that the aggregate signal never goes below the minimum value specified for each power state.

6.3.10. Parameters

The are several important `const` parameters defined throughout the code[3] which affect the performance. Let us quickly discuss some of the more important parameters. (The current value will also be given)

`PowerStateGraph::update() TOP_SLICE_SIZE = 10` The number of spikes to take from each signature. A larger value tends to produce a more complex (and hence more constrained) power state graph. When training with two washing machine signatures, setting `TOP_SLICE_SIZE` to

[3] I would prefer to let users set these parameters at runtime in `disaggregate.conf` but this was not a priority during development so, at the current stage of development, these parameters are set inside the code.

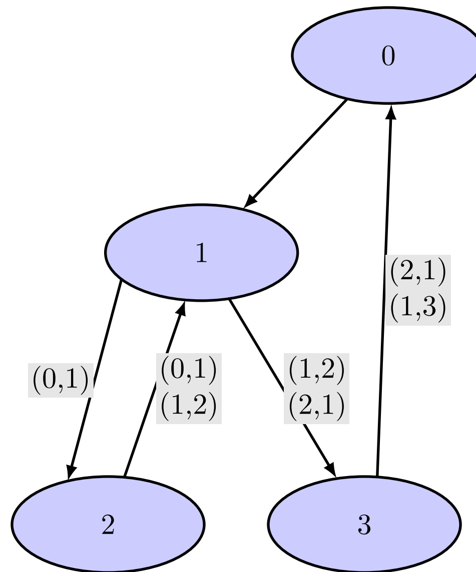


Figure 6.10.: A power state graph with an edge history length of 2.

10 produces a power state graph with 4 vertices and 9 edges whilst setting `TOP_SLICE_SIZE` to 100 produces 5 vertices and 12 edges (it does still disaggregate correctly).

`PowerStateGraph::EDGE_HISTORY_SIZE = 5` The length of the "rolling edge history" (see section 6.3.9). Setting this to 0 disables the edge history which has no effect on the disaggregation performance for simple devices like kettles but makes the system take many minutes to disaggregate complex devices with cyclic power state graphs. Setting `EDGE_HISTORY_SIZE` higher than `TOP_SLICE_SIZE` effectively turns the power state graph into a flat list.

`PowerStateGraph::mostSimilarVertex() ALPHA = 0.0000005` The significance level for the t-test (what constitutes as a "satisfactory" match?)

`PowerStateGraph::traceToEnd() WINDOW_FRAME = 8` This is the number of seconds by which to widen the search window (i.e. `WINDOW_FRAME` is subtracted from `begOfSearchWindow` and added to `endOfSearchWindow`). What is the purpose of this parameter? Recall that the aggregate data has a sample period of 6 seconds whilst the signature data has a sample period of 1 second, and that the Current Cost smart meter sometimes drops samples. This means that a power state duration learnt from a signature is unlikely to be represented precisely in the aggregate data. Let me illustrate the problem by example: say the disaggregation algorithm is expecting a spike at, say, 00:00:10 but the two nearest samples in the aggregate data are at 00:00:04 and 00:00:16. The job of `WINDOW_FRAME` is to widen the search window to minimise the chance of missing an event in the aggregate data due to the sample-rate mismatch.

`AggregateData::findSpike()` There are several parameters within this function for tweaking how strict `findSpike()` is while looking through $\Delta aggregate$ for a specific spike.

6.3.11. Testing and debugging

Units tests were implemented using the Boost Test Library[4]. Whenever possible, tests were written at the same time as the function under test. The full suite of tests can be run with `make testAll`. Alternatively, individual tests can be run with `make <testtarget>` where `testtarget` is one of the following: `ArrayTest, GNUplotTest, UtilsTest, StatisticTest, PowerStateGraphTest, AggregateDataTest`.

[4] boost.org/doc/libs/1_42_0/libs/test/doc/html/index.html

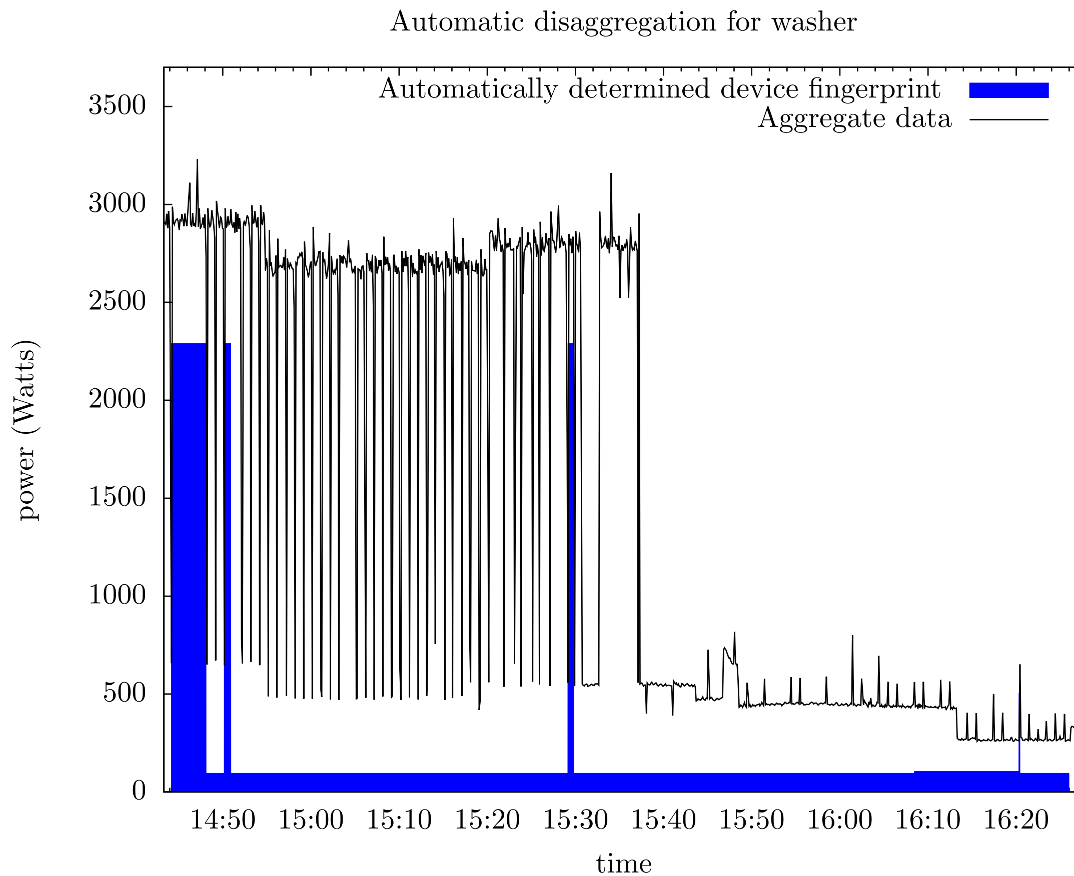


Figure 6.11.: Incorrect disaggregation. The disaggregation algorithm has attempted to fit a powerStateGraph trained on washing machine signatures to a section of aggregate data dominated by the tumble drier. Note that, in places, the aggregate signal actually dips into the fingerprint power state.

One of the biggest challenges during the software development was testing and debugging the signal processing functions. Testing simple little functions like like `bool Utils::roughlyEqual (double, double)` was easily done using routine unit testing but testing larger functions like `PowerStateGraph::disaggregate()` was much harder. Detecting and fixing *obvious* problems like `seg faults` was not a problem. The challenge came when the code ran fine but produced *slightly* wrong answers. In an attempt to make it easier to test the signal processing functions, a handful of synthetic signatures and aggregate data files were created. The real headaches came when the code worked perfectly on all the tests but failed on *real* data. I found `gdb` not particularly useful in this situation because the bug might be inside a loop which executes 1,000 times and might fail on just one of those iterations (it's entirely possible that I've missed a clever feature of `gdb` which would have helped).

For a while the code made use of Google's Logging Framework "glog"[5] to output diagnostic logging information about every function but this log file soon became unwieldy so glog was removed from the code.

I settled on three broad solutions for running diagnostics. The first is to make extensive use of `GNUplot` (via my GNUplot wrapper functions - see chapter 5) for plotting signals and `graphviz`[6] for plotting graphs. It is crucial to be able to visualise each algorithm's data input and output as quickly as possible. If you can't see the data then you're developing "blind".

The second solution is just standard programming practice: many functions start by doing a lot of "sanity checking" to make sure they are getting the data they expect.

[5] code.google.com/p/google-glog
[6] `graphviz.org`

The third solution is to simply print diagnostic information to the standard output. Each hard-to-debug function takes a `bool verbose` argument (which defaults to `false`). If this is set to `true` then the function prints diagnostic information to the standard output which can be piped to a text file for analysis. It is not sufficient to have a single global `DEBUG` flag because it is often necessary to inspect the behaviour of just a single function. The single biggest problem with this approach is that it clutters up the code with lots of `if (verbose) cout ...` statements.

6.4. Performance

6.4.1. `10July.csv` aggregate data

During the design and debugging I used 24 hours of aggregate data (`data/input/current_cost/10July.csv`) and two washer signatures, one toaster signature and two kettle signatures. One of each device signature was recorded during this 24 hour period. Let us see how the system performs on this 24 hour period of aggregate data.

I will state the commands used to run each test. But first, here is a very quick tutorial on using `disaggregate` (a more detailed manual is in the appendix): The main way to use `disaggregate` is:

```
./disaggregate [AGG_DATA_FILE] -s [SIG_FILE] -s [ANOTHER_SIG_FILE] -n [DEVICE_NAME]
```

e.g.:

```
./disaggregate 10July.csv -s toaster.csv -s toaster2.csv -n toaster
```

The path for the aggregate data file is `data/input/current_cost` and the path for the signature data is `/data/input/watts_up`

Please note that each row in the following data tables represents an actual device activation (i.e. a device activation which we know for certain did happen). A "false negative" indicates that the disaggregation algorithm failed to detect an event.

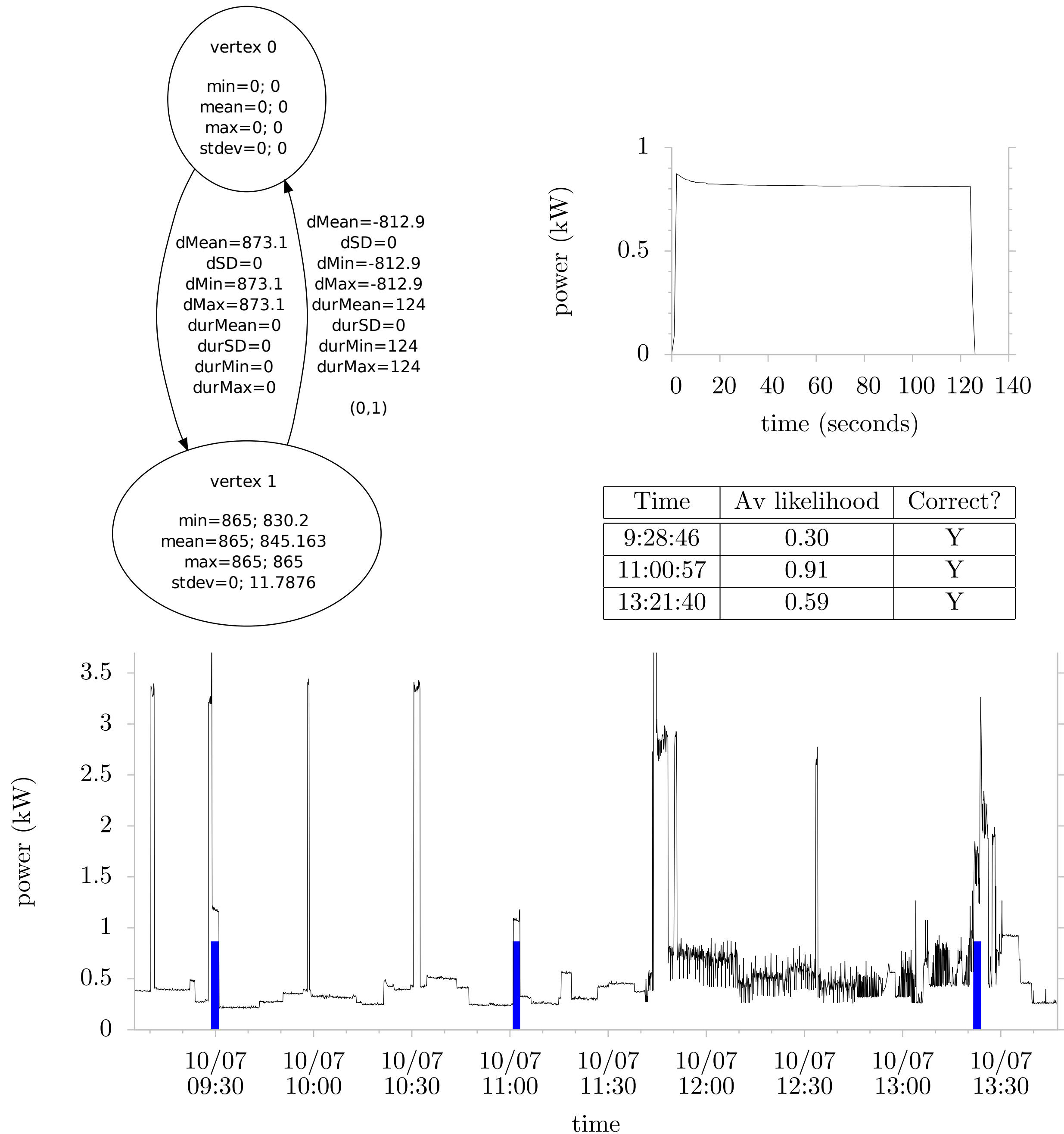
In all the disaggregation graphs; the solid blue blocks mark the output of the disaggregation algorithm and the black line marks the aggregate signal.

The power state graphs shown below show a lot of information. Each vertex shows four rows of data (min, mean, max, stdev) and two columns (the power state stats determined from the 8 samples immediately after each spike; the power state stats determined from every sample between adjacent spikes). Each edge shows a set of stats prefixed with a "d"; these are the statistics describing the **d**elta. Each edge also shows a set of stats prefixed with "dur"; these are the **dur**ation stats (in seconds). Each edge also shows its edge history.

Toaster

Trained on `toaster.csv` (which was recorded simultaneously with the aggregate data)

```
./disaggregate 10July.csv -s toaster.csv -n toaster
vertex1 = {min=830.2 mean=845.163 max=865 stdev=11.7876 numDataPoints=8}
```



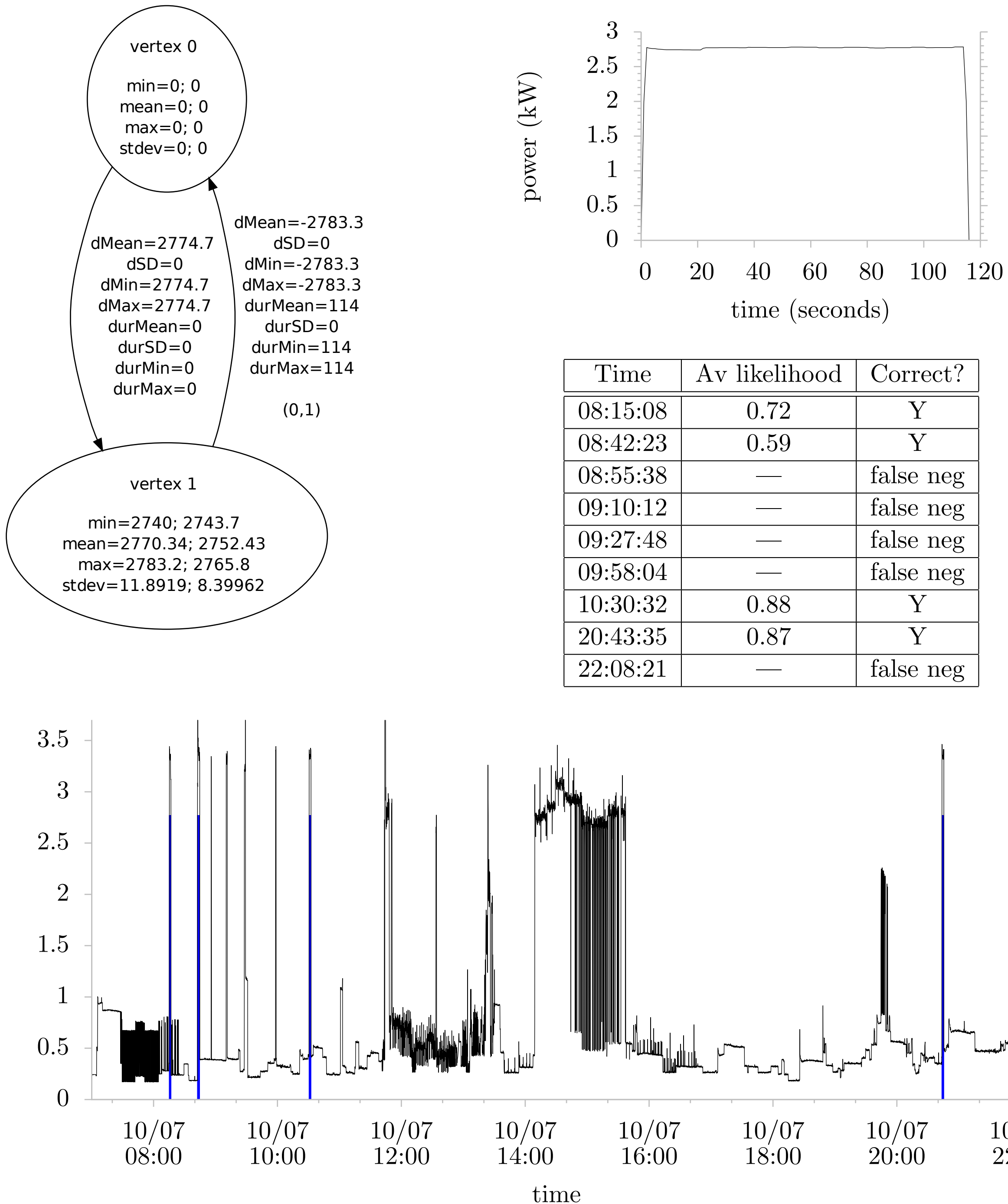
Time	Av likelihood	Correct?
9:28:46	0.30	Y
11:00:57	0.91	Y
13:21:40	0.59	Y

Figure 6.12.: The `toaster.csv` signature file is shown in the top right panel. The graph on the top left is the power state graph determined from the single `toaster.csv` signature file. The table to the right of the power state graph shows three rows representing three actual activations of the kettle and showing that the disaggregation algorithm found all three toaster activations. The graph at the bottom of the page shows the aggregate data in black and the automatically determined activations in blue. The `toaster.csv` signature was recorded at 11am hence the high likelihood for the 11am event.

Kettle

Trained on only `kettle.csv` (which was recorded simultaneously with the aggregate data)

```
./disaggregate 10July.csv -s kettle.csv -s kettle2.csv -n kettle
vertex1 = {min=2743.7 mean=2752.43 max=2765.8 stdev=8.39962 numDataPoints=8}
```



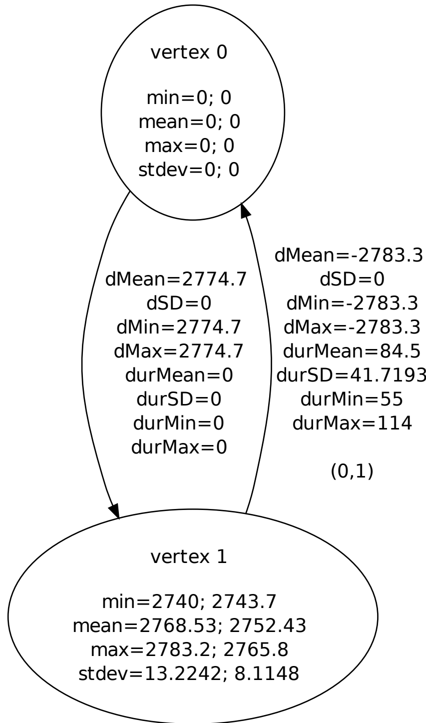
Time	Av likelihood	Correct?
08:15:08	0.72	Y
08:42:23	0.59	Y
08:55:38	—	false neg
09:10:12	—	false neg
09:27:48	—	false neg
09:58:04	—	false neg
10:30:32	0.88	Y
20:43:35	0.87	Y
22:08:21	—	false neg

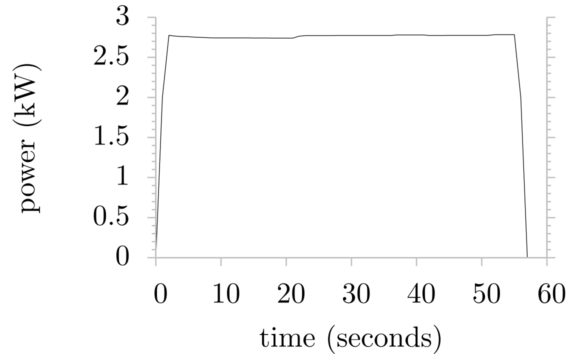
Figure 6.13.: Top left: power state graph trained on `kettle.csv`. Top right: `kettle.csv` signature. Table: disaggregation performance. Bottom: the solid blue plot shows where the disaggregation algorithm believes there's a kettle fingerprint; the black line is the agg. signal.

Note that the system fails to detect 5 out of 9 actual activations of the kettle. This is simply because there is a wide range of *durations* for the kettle yet the edge on the power state graph from vertex 1 to 0 states a "durSD" (standard deviation of the power state duration) of 0 because the PSG has been trained on only a single signature. Perhaps the system will do better if we train with 2 kettle signatures (to better represent the full range of possible kettle durations)...

Trained on both `kettle.csv` and `kettle2.csv`

```
./disaggregate 10July.csv -s kettle.csv -s kettle2.csv -n kettle
vertex1 = {min=2743.7 mean=2752.43 max=2765.8 stdev=8.1148 numDataPoints=16}
```





Time	Av likelihood	Correct?
08:15:08	0.89	Y
08:42:23	0.64	Y
08:55:38	0.50	Y
09:10:12	0.87	Y
09:27:48	0.90	Y
09:58:04	0.55	Y
10:30:32	0.75	Y
20:43:35	0.79	Y
22:08:21	0.83	Y

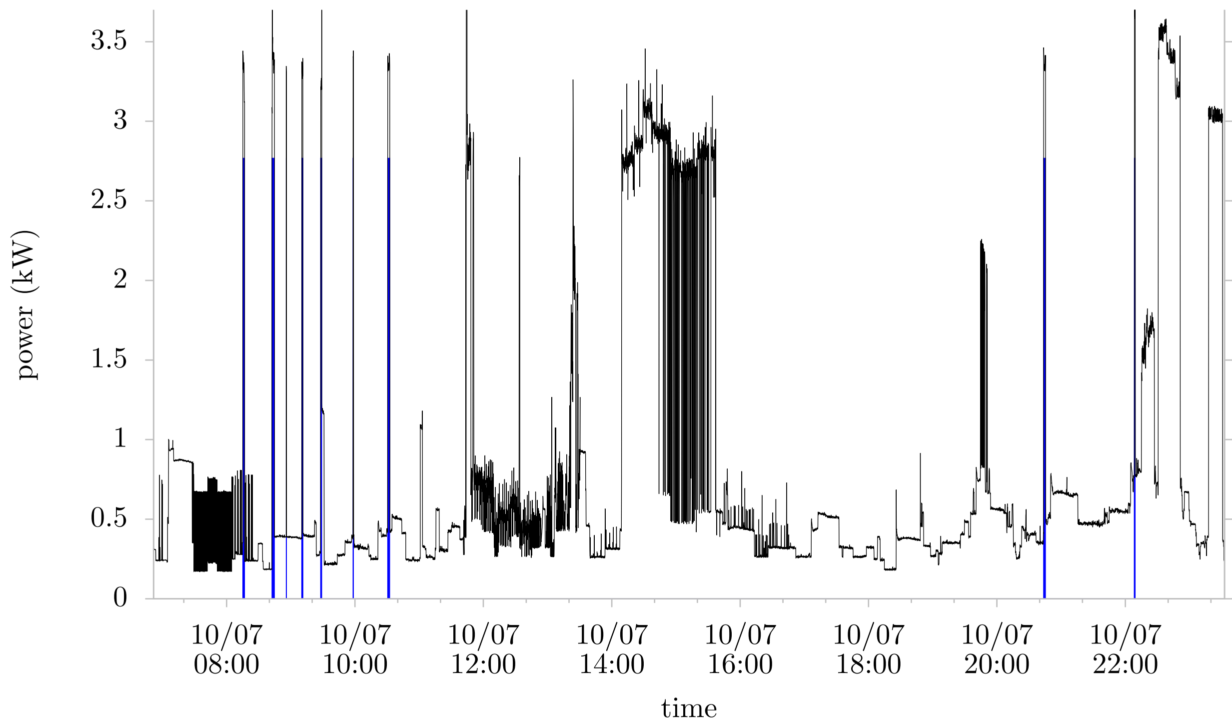


Figure 6.14.: Top left: power state graph trained on `kettle2.csv` and `kettle.csv`. Top right: `kettle2.csv` (half the duration of `kettle.csv`). Table & bottom graph: disaggregation.

The system disaggregates the kettle perfectly now that it has been trained using both kettle signatures. Note that the edge on the power state graph from vertex 1 to 0 states a “durSD” (duration standard deviation) of 41.7 reflecting the fact that the two signatures have a different duration.

Washer

Trained on just `washer2.csv` (which was recorded simultaneously with the aggregate data)

```
./disaggregate 10July.csv -s washer2.csv -n washer
vertex1 = {min=2265.1 mean=2331.2 max=2396.8 stdev=21.54     numDataPoints=24}
vertex2 = {min=   2.5 mean= 140.1 max= 282.7 stdev=72.75     numDataPoints=32}
vertex3 = {min=  14.7 mean=  14.7 max=  14.7 stdev= 1.89e-15 numDataPoints= 8}
vertex4 = {min= 468.7 mean= 517.0 max= 553.7 stdev=33.18     numDataPoints= 8}
```

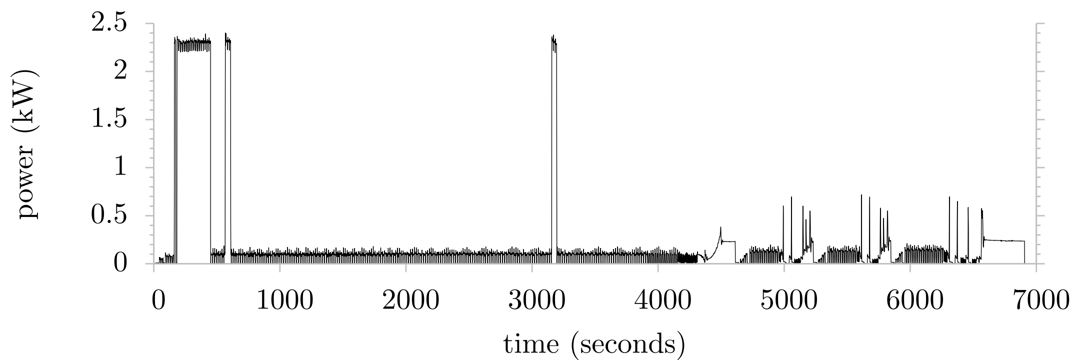


Figure 6.15.: `washer2.csv` signature

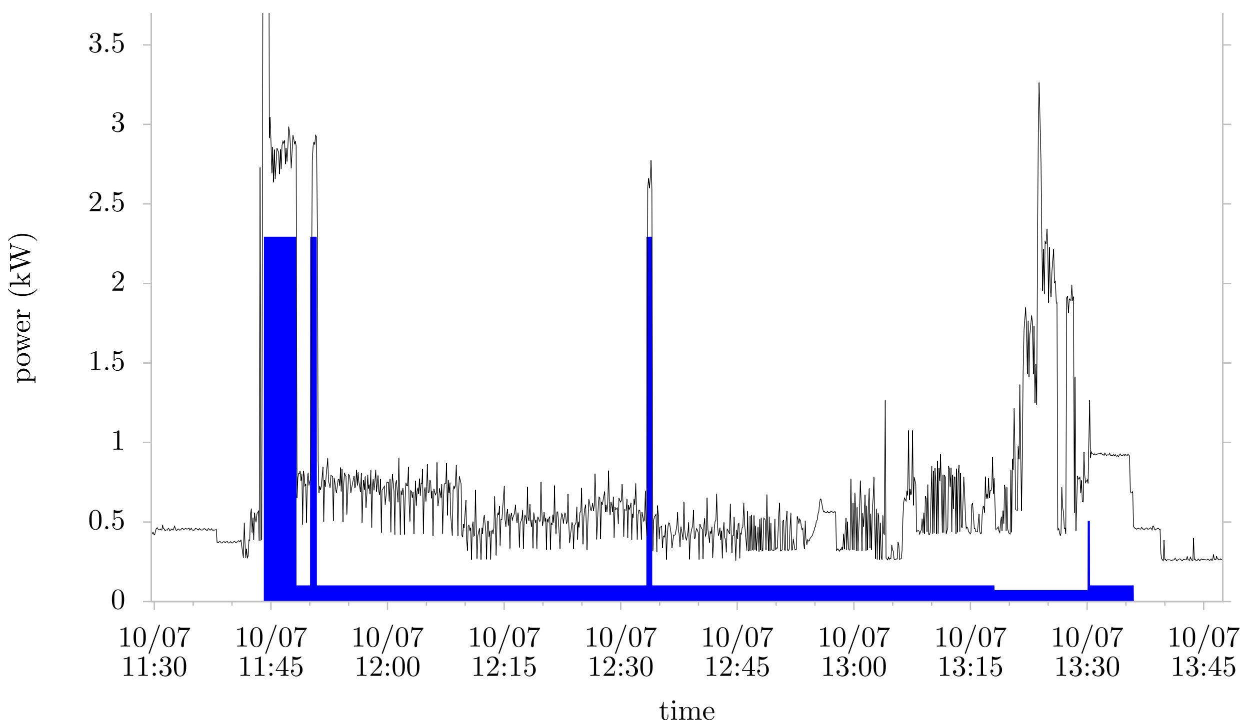


Figure 6.16.: The solid blue shows the disaggregation algorithm's estimation of where the washer fingerprint is. It has successfully located the main elements of the washer signature.

Time	Av likelihood	Correct?
11:41:06	0.88	Y

The power state graph is on the next page.

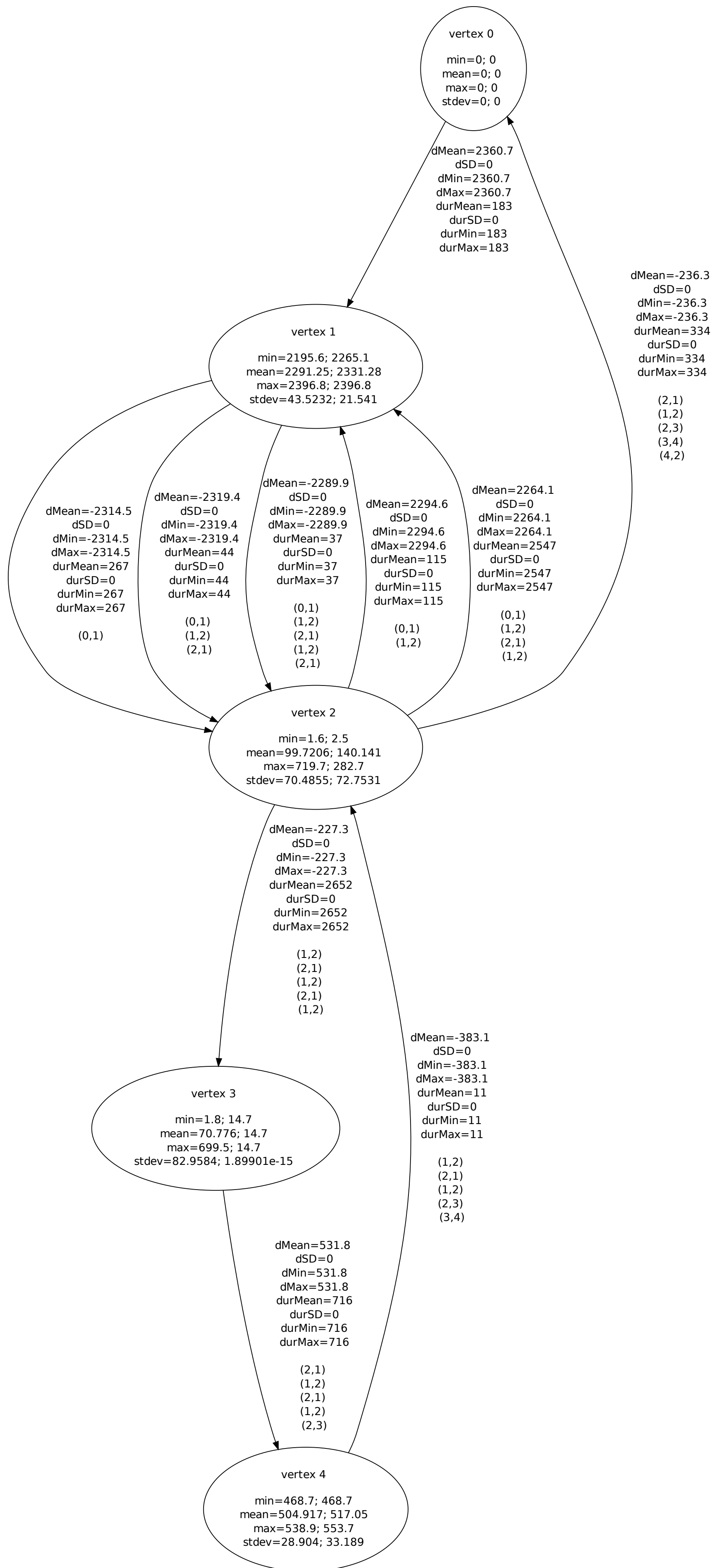


Figure 6.17.: Power state graph trained on `washer2.csv`

Trained on `washer2.csv` and `washer.csv`

```
./disaggregate 10July.csv -s washer2.csv -s washer.csv -n washer
vertex1 = {min=2239.3 mean=2325.0 max=2396.8 stdev=32.1 numDataPoints=40}
vertex2 = {min=   2.5 mean= 127.5 max= 282.7 stdev=62.2 numDataPoints=48}
vertex3 = {min=  14.3 mean=  14.5 max=  14.7 stdev= 0.2 numDataPoints=24}
vertex4 = {min= 468.7 mean= 517.1 max= 553.7 stdev=33.2 numDataPoints= 8}
```

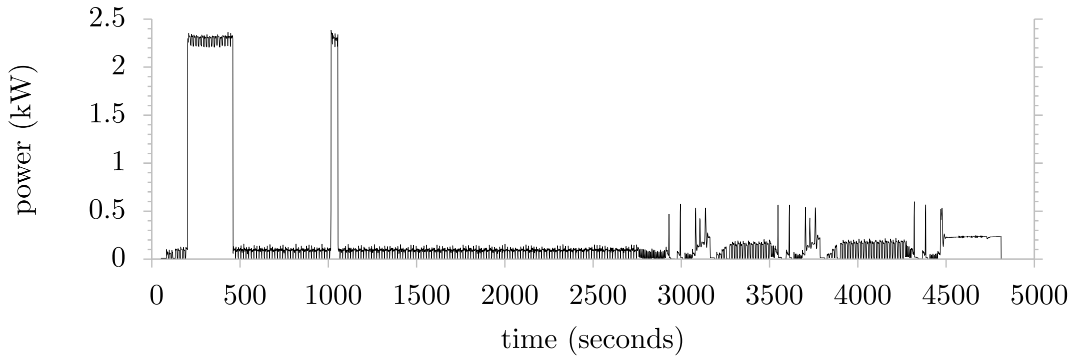


Figure 6.18.: `washer.csv` signature. Note that the heater only comes on twice in this signature, whilst it comes on three times in `washer2.csv` (figure 6.15).

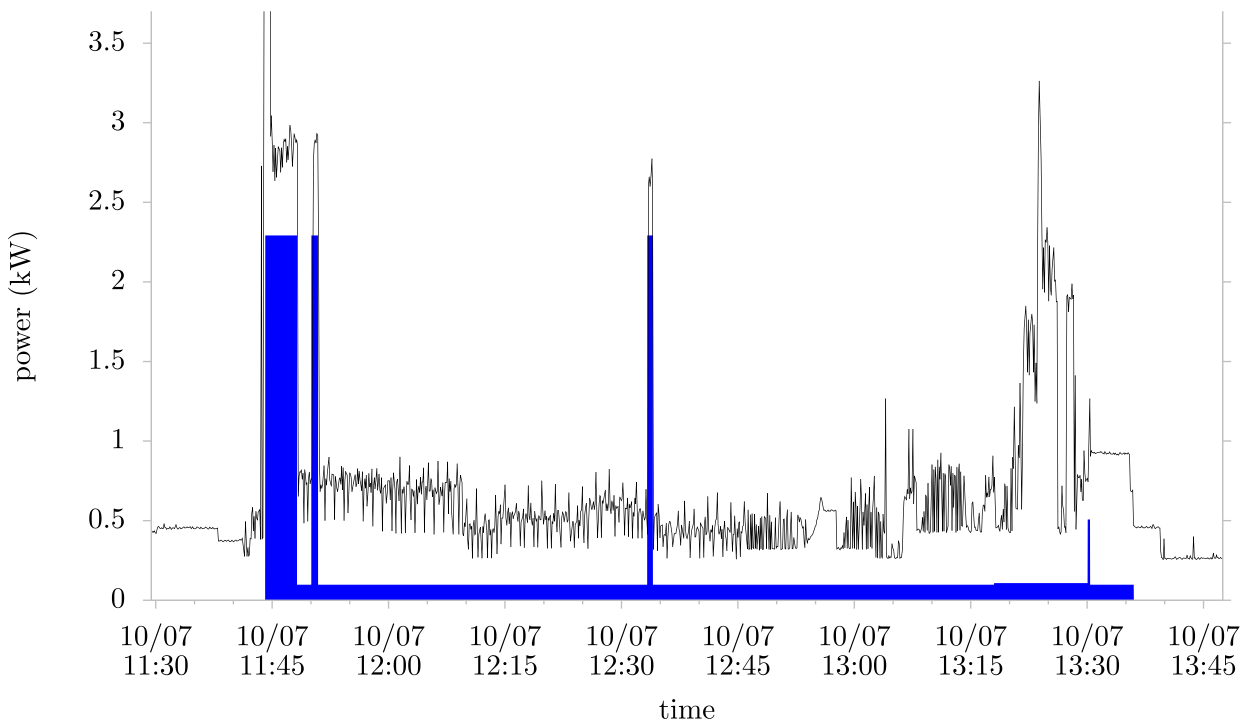


Figure 6.19.: The solid blue shows the disaggregation algorithm's estimation of where the washer fingerprint is. It has successfully located the main elements of the washer signature. This disaggregation output is almost identical to the output when trained with only `washer2.csv` (figure 6.16), despite the significantly more complex power state graph produced by trained on both `washer2.csv` and `washer.csv` (shown on next page).

Time	Av likelihood	Correct?
11:40:56	0.80	Y

The power state graph is on the next page.

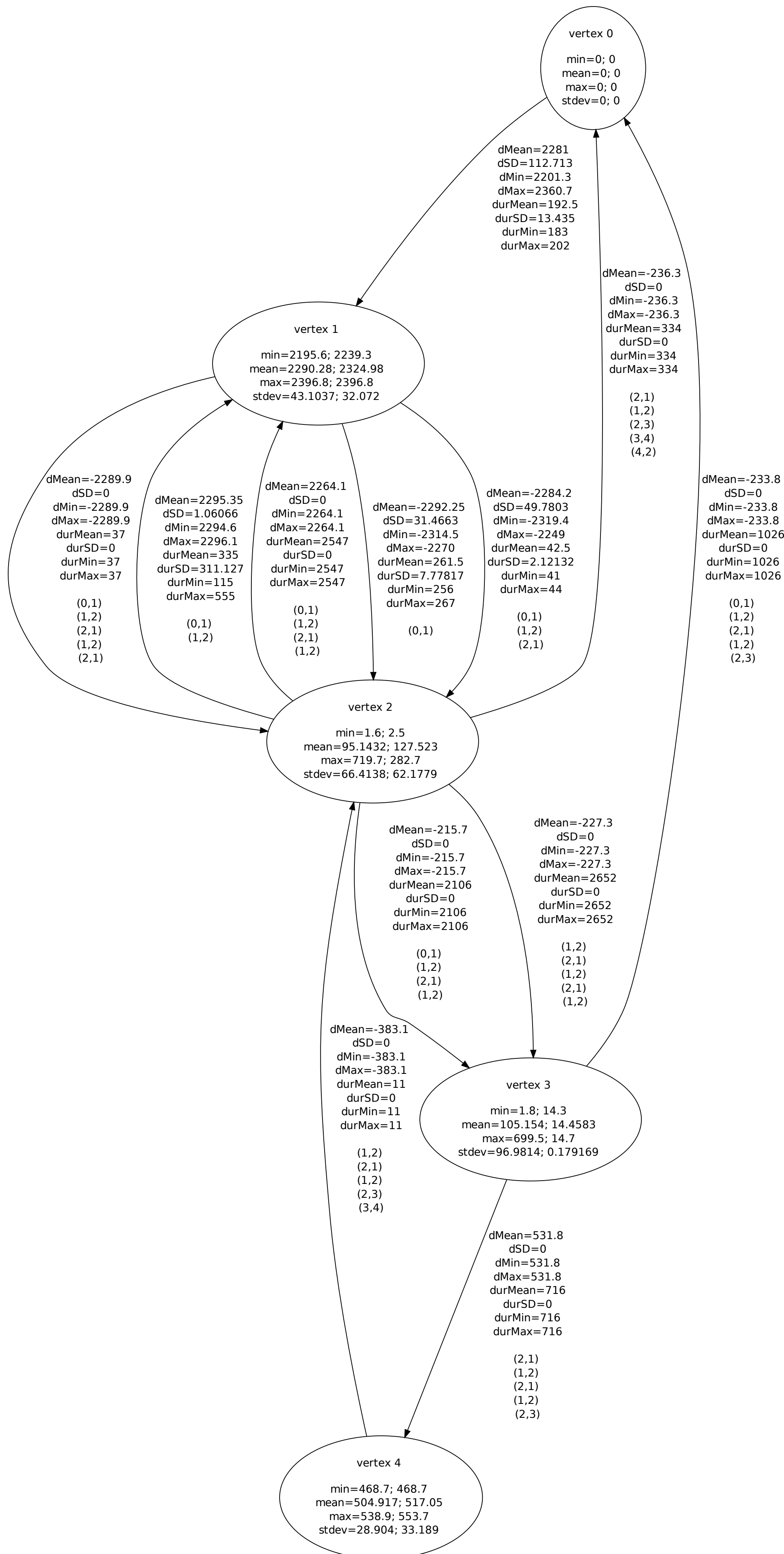


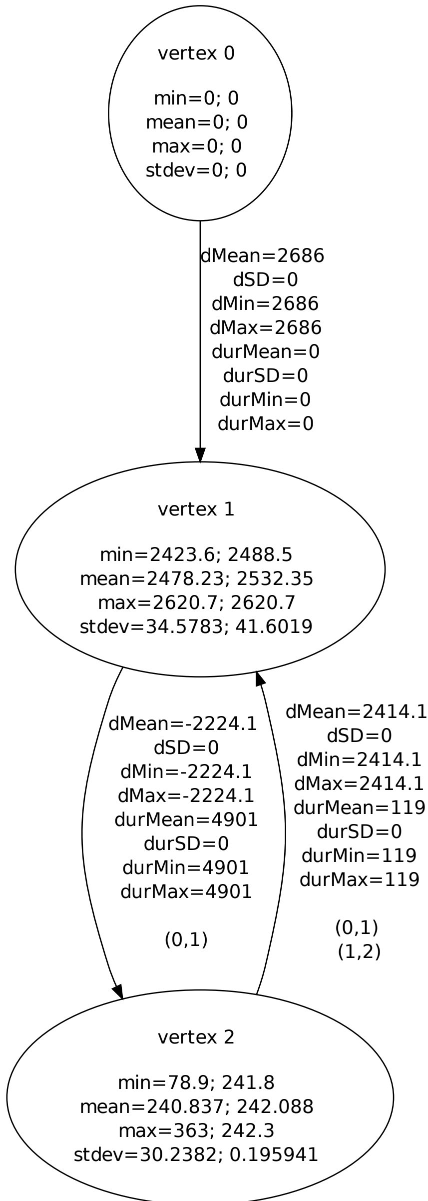
Figure 6.20.: Power state graph trained on `washer.csv` and `washer2.csv`

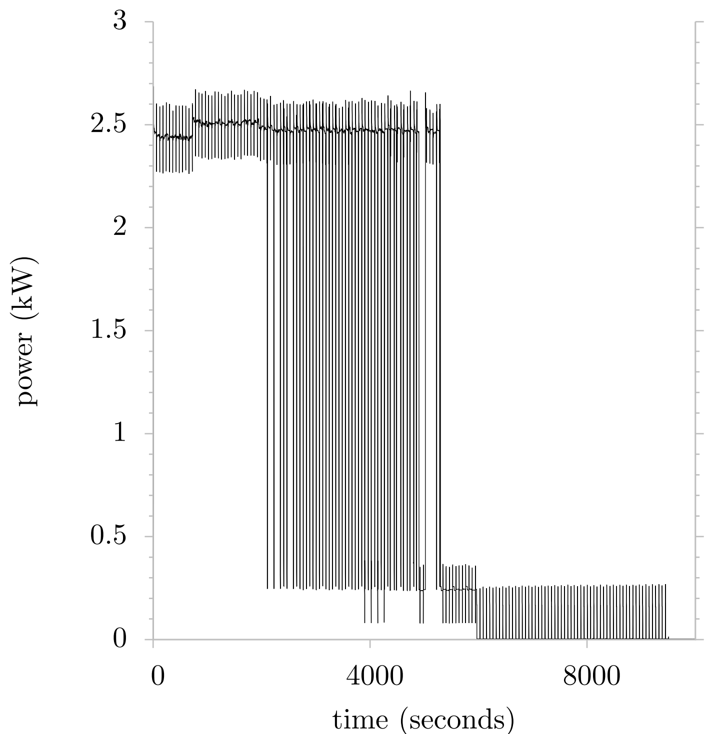
Tumble drier

Trained on just `tumble.csv`

```
./disaggregate 10July.csv -s tumble.csv -n tumble
vertex1 = {min=2488.5 mean=2532.6 max=2620.7 stdev=41.6 numDataPoints=16}
vertex2 = {min= 241.8 mean= 242.1 max= 242.3 stdev= 0.2 numDataPoints=8}
```

Time	Av likelihood	Correct?
14:10:00	—	False neg





The system has failed to represent the signature (shown on the right) as a power state graph (shown on the left) and hence failed to detect the tumble drier's fingerprint in the aggregate data. Note a fundamental flaw in this power state graph: the off vertex lacks an inbound edge. Apparently the tumble drier never turns off! This is almost certainly because the spike which represents the transition back to the offvertex did not make it into the top 10 spikes (defined by the parameter TOP_SLICE_SIZE) because, as can be seen in the signature above, the tumble drier has many, many large spikes.

6.4.2. 3 days of aggregate data (`earlyJuly.csv`)

This is data was not been used during development. Instead of providing a line-by-line account, just the scores will be reported. In all cases, the algorithm will be trained on every signature file available (2 for the kettle, 1 for the toaster, 5 for the washing machine and 1 for the tumble drier).

	% hits	failed to detect	false positives
Toaster	100 % (6 out of 6)		
Kettle	100 % (28 out of 28)		
Washer	100 % (4 out of 4)		1
Tumble	0 % (0 out of 2)	2	

6.4.3. 10 days of aggregate data (`earlyAugust.csv`)

As before, in all cases, the algorithm will be trained on every signature file available (2 for the kettle, 2 for the toaster, 5 for the washing machine and 1 for the tumble)

	% hits	failed to detect	false positives
Toaster	100 % (5 out of 5)		
Kettle	100 % (25 out of 25)		6
Washer	100 % (2 out of 2)		6
Tumble	0 % (0 out of 3)	3	

6.4.4. Conclusions

Broadly, I'm pleased with the performance of the system. The "power state graph" approach does appear to be a successful strategy for abstracting raw device signatures and for locating those signatures in aggregate data.

The algorithm worked very swiftly (within around 1 second) for all disaggregation tasks except the final washing machine disaggregation task for the 10-day-aggregate-data (which took 20 minutes!). It appears that this run was very long because the algorithm fails to correctly determine how `washer5` finishes, probably because the spike which indicates a transition to "off" does not come within the top 10 spikes for `washer5`. Increasing `TOP_SLICE_SIZE` (section 6.3.10) fixes this somewhat but this is definitely not the correct behaviour.

Overall, the system does appear capable of learning a "power state graph" from several signatures and of locating those device fingerprints within aggregate data.

7. Conclusions and future work

7.1. Limitations

Complete failure to deal with devices with rapidly changing signatures. The disaggregation algorithm works well for the kettle, toaster and washer but fails for the tumble drier. The tumble drier causes yet more headaches: In the 10 day aggregate data, all the false positives for the kettle and for the washer were all within tumble drier fingerprints (the tumble drier's heater uses a similar amount of power to the kettle; see figure 7.1). Supposedly this is because the drier has a very "choppy" signature (figure 7.1) which confuses the training algorithm (recall that the training algorithm creates statistics for the 8 samples immediately before and after each "$\triangle signature$ spike" and rejects a spike if these "pre- / post-spike statistics" have an unacceptably high standard deviation). This is a serious flaw, not least because tumble driers are amongst the most energy-hungry devices in the home (a single run uses around 3.5 kWh).

`PowerStateGraph.cpp` and `.h` are large files (approximately 1,200 and 400 lines respectively). The `PowerStateGraph` class could be broken into multiple smaller classes. In particular, the code for training a power state graph and the code for disaggregating an `AggregateData` signal from this graph could be separated into 2 classes: a `PowerStateGraph` class for *creating* a power state graph and a `Disaggregate` class which wraps a "disaggregation tree". The "rolling edge history" could potentially also be its own class.

"Edge histories" are a fairly blunt instrument for constraining the power state graph. A better alternative may be to implement a system which only limits the number of times the disaggregation algorithm is permitted to spin round *cyclical* sections of the power state graph.

This approach will never be able to disaggregate the "vampire power" load. "vampire power" is the baseline that a building uses when the house appears to be "doing nothing". This load is typically around 50-200 Watts, continually drawn every hour of the day, every day of the year. Because this load is constant, the total contribution to yearly energy consumption is often quite high (maybe around 20 %). This load is largely attributable to devices which continue to draw power even when in standby and devices which genuinely never turn off, like alarm systems. It would be very useful to know exactly which devices contribute to the vampire power load. Unfortunately, none of the disaggregation techniques attempted during this project have any hope of disaggregating the vampire power load. The vampire load is always on, so it never produces a "spike in $\Delta aggregate$", so it is invisible to my disaggregation algorithm.

This approach is likely to struggle with loads which vary continuously. For example, this approach is unlikely to be able to disaggregate a dimmable lamp.

7.2. Other applications of this work

The amount of energy consumed by IT infrastructure is growing rapidly. It would be interesting to see if this disaggregation algorithm would be able to determine how much power is consumed by each component in a computer or a data centre.

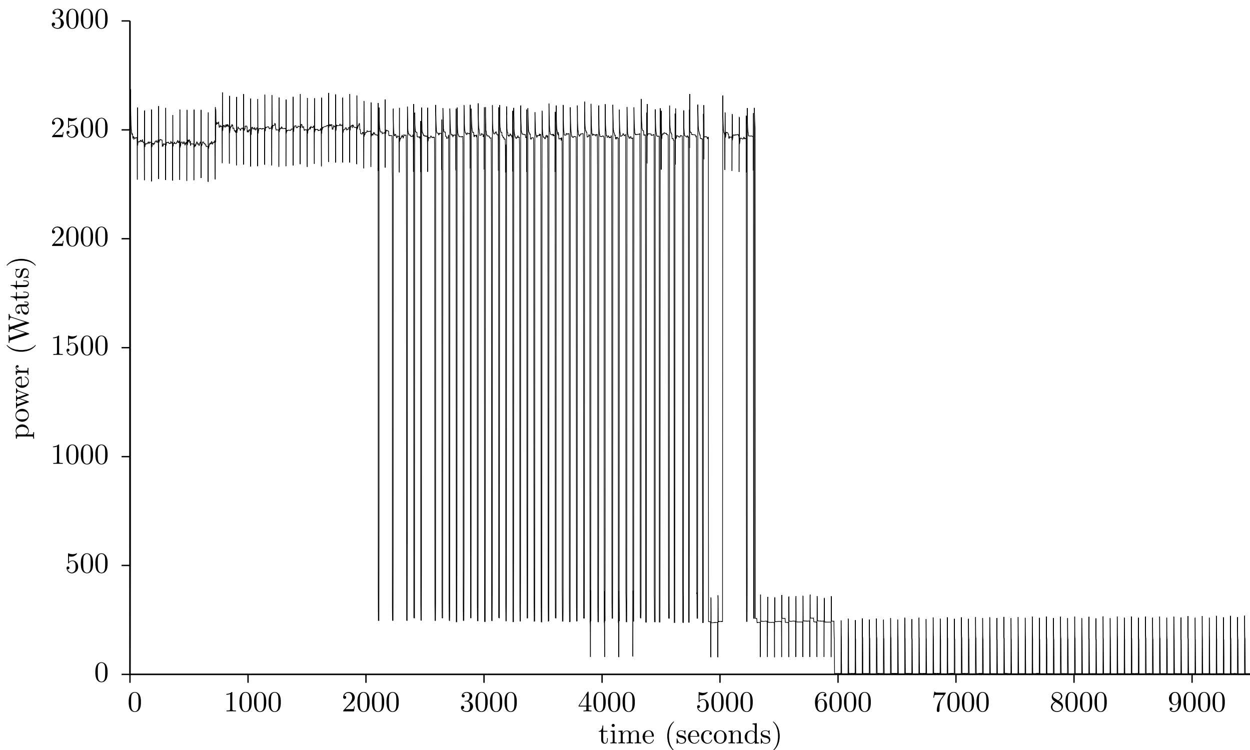


Figure 7.1.: Tumble drier signature

7.3. Further work

7.3.1. Combine all three approaches

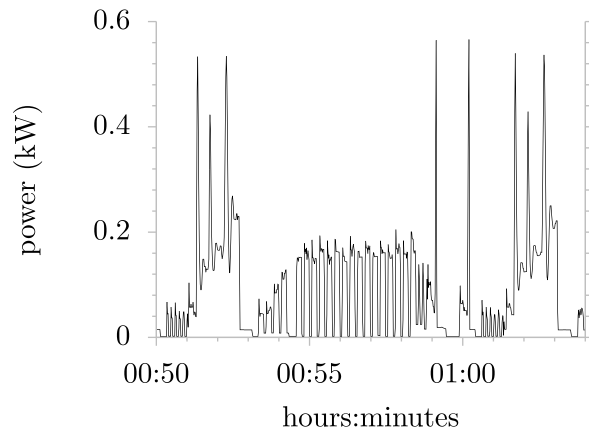


Figure 7.2.: Cropped washing machine signature showing two interesting "ramps" at 0:51 & 1:02

Three broad approaches were explored during this project (1. least mean squares, 2. determining "power states" by examining the signature's histogram and 3. the "graphs and spikes" approach). These approaches could be combined. For example, the "graphs and spikes" approach completely fails to take advantage of highly discriminative features like the interesting "ramps" in the washing machine signature (figure 7.2). Perhaps a least mean squares approach could be deployed for short waveforms like this ramp. It should be relatively straight forward to build an LMS algorithm which is robust against background noise by using the LMS algorithm to match a waveform in *sections* (this idea was suggested by my supervisor, Dr Knottenbelt).

7.3.2. Machine learning approaches

Most modern cameras can recognise faces and distinguish between individuals. This "face recognition" problem appears to be a considerably harder problem than disaggregation problem. It may be interesting to survey existing machine learning algorithms and attempt to apply them to the disaggregation problem.

7.3.3. More efficient implementation & parallelisation

The focus during this project has been to develop a set of algorithms. My priority when writing the code was to create code which works robustly. As such, many of the algorithms are not optimally implemented. For example, several functions do a lot of sanity-checking which would not be necessary once the code is fully debugged.

All the code is single-threaded. There are many opportunities for using multiple CPU / GPU threads or single-instruction-multiple-data (SIMD) instructions.

7.3.4. Try a simpler approach based on non-invasive load monitoring

We discussed the "non-invasive load monitoring" (NILM) algorithm in the Introduction. The NILM algorithm was developed for use where measurements of *real* and *reactive* power are available and has not, to my knowledge, been applied to the disaggregation of home energy monitor data (which lacks information about *real* versus *reactive* power).

The NILM algorithm starts by looking for changes in the aggregate data and then pairs changes of equal magnitude but opposite sign. The very last step of the NILM algorithm is to compare these pairs of changes to a database of learnt device signatures. In contrast to the NILM approach, my disaggregation algorithm looks for learnt signatures as the first and only thing it does; the NILM algorithm instead simply looks for equal but opposite spikes first and only then accesses its database of learnt signatures. A more literal port of the NILM algorithm to home energy monitor data may be more efficient. However, it may also be too poorly constrained to cope with the relatively course data provided by home energy monitors. The advantage of my "graph and spikes" approach is that the disaggregation algorithm only looks for events which are relevant to the device currently being disaggregated.

7.3.5. Remove deprecated sections of code

This project has been an experiment and, as such, the design has evolved over the course of the code's construction. I deliberately did not remove functions belonging to previous design iterations for two reasons: firstly, when I started out on a new design modification, I could not be sure that it would work better than the previous design iteration. I did start a new git branch for the "Spikes and Graphs" design but this has just become my "master" branch. Secondly, I wanted to keep all my work in a single code base so I could refer to functions from previous design iterations in the write-up.

The end result is that a fair amount of the current code-base is simply not used by the "Spikes and Graphs" design, hence the code could be slimmed down.

7.3.6. Capture relationships between devices

Many manufacturers produce household devices. Whilst it's likely that every kettle[1] behaves almost identically, it's highly likely that devices like washing machines vary greatly in their behaviour. The disaggregation system should be able to capture the fact that each washing machine is a member of the same class of device.

7.3.7. Capture the probability that a device is active at a given time of day

Kettles and toasters tend to be used at meal times; TVs tend to be used in the evening; fridges run constantly. The disaggregation system could record a probability that each device will be on at a specific time. This probability would be used to refine the disaggregation system's *likelihood* calculation.

[1] Assume we're only considering devices made for a 230 Volt, 13 Amp domestic mains supply like the UK's. The UK's domestic mains supply can deliver up to 2.9 kW($= 230\,V \times 13\,A$) to each device whilst a North American 120 Volt, 15 Amp socket can deliver a maximum of 1.8 kW. British kettles are more powerful than American kettles!

7.3.8. Ultimate aim: to build a smart meter disaggregation web service

The ultimate aim is to develop this software into a web service which would work something like this: users would send the service their aggregate signal as a flat data file and the service would return an XML file listing which device signatures were found in the aggregate signal, their start times, duration and energy consumption. Users could then use this information to help them manage their energy consumption.

Disaggregated domestic energy information should be considered private information because it describes when you're at home, when you're watching TV, when you're having a power shower, when you get up in the morning, when you go to bed etc. So the transfer of data to and from the disaggregation service must be encrypted and secure.

If the system was thoroughly optimised and parallelised then the web service could be run on a single server while the user base was small, or perhaps it could be run on an elastic computing service like Amazon's EC2 service[2].

There are several reasons why it might be nice to put a disaggregation service "in the cloud":

A single, shared library of device signatures. The first handful of users would have to invest time in establishing a shared library of device signatures, but those users would see instant personal benefit from doing so because their work will result in the service being more useful to them. Once the library covers the majority of household devices, new users would not have to train the system, they can use the service simply as consumers.

Energy information already exists in the cloud. Many energy-related web services and Internet-enabled devices already exist. Hence transmitting energy data to the disaggregation service will, in some cases, be as simple as writing a small utility to ingest data from an existing service. For example, Pachube.com (pronounced "patch bay") calls itself a "*Real-Time Open Data Web Service for the Internet of Things*". It offers a way to "patch" sensors to actuators across the Internet. It also offers tools to produce live graphs of sensor data. Some people already send their smart meter data to Pachube. The Current Cost Bridge "bridges" the Current Cost smart meter to a wired Ethernet Internet connection to automatically transmit smart meter data[3] to Current Cost's web service (using Pachube for backend data management [1]). Another smart meter called the "AlertMe"[4] transmits data to AlertMe's website roughly every 10 seconds; AlertMe provide an API for access to the data.

How much energy do similar appliances use? At the very start of this report, we considered a consumer who has just received a painfully expensive electricity bill. She might find it especially useful to know, for example, that her ageing fridge uses x % more energy than the average fridge and hence a new fridge will pay for itself in y months. Providing a disaggregation service in the cloud would allow this type of comparative data to be collected easily.

How much energy do my friends use? With a disaggregation service in the cloud, it would be possible to make smart meter disaggregation data "social" and to turn energy saving into a social game (of course, users would require complete control over which friends see what data). Users could compare their energy usage with their friends' usage and compete to see who can save the most energy.

[2] aws.amazon.com/ec2

[3] Unfortunately the Current Cost Bridge does not transmit the 6-second resolution data available from the Current Cost's USB connection. Instead the bridge sums and averages the 6-second data over a 5 minute period and transmits this sum and average every 5 minutes [2]. This makes the bridge's data less useful for disaggregation than the 6-second data available from the Current Cost's USB connection. However, there are plenty of alternative ways to transmit the Current Cost data to the Internet (e.g. github.com/Floppy/currentcost-daemon or community.pachube.com/currentcost).

[4] alertme.com/products/energy

7.4. Conclusion

Let us summarise what has and what has not been achieved in this project. On the down side, the system fails to learn a valid representation for the tumble drier which is unfortunate because the drier is one of the most power-hungry domestic devices. On the positive side, we have designed and implemented a novel disaggregation system which successfully learns valid representations for three out of the four tested devices. The system rapidly and successfully disaggregates these three devices from noisy aggregate data.

This design has several attractive features: 1) it handles complex devices whose signatures contain power state sequences which repeat an arbitrary number of times; 2) it is probabilistic; 3) a power state graph can be learnt from one or more signatures and 4) the design estimates the energy consumed by each device activation.

Has the project met the original aims? Whilst this is not production-ready system, it is a productive contribution towards that goal. The original aim was to design and build a working disaggregation system, which we have achieved.

A. User guide

The system described in this report is implemented as a command-line utility called `disaggregate`. An x86-64 Linux executable is included with the submitted archive and should run "out of the box" provided the runtime dependencies listed below are available (`disaggregate` compiles and runs fine on the DoC lab machines). Running `./disaggregate` without any arguments will produce the standard help:

```
$ ./disaggregate
SMART METER DISAGGREGATION TOOL
Signature file(s) must be specified using the -s option.
Usage: ./disaggregate AGGREGATE_DATA_FILE -s SIGNATURE [OPTIONS]
Learns the salient features of a single device
 from the SIGNATURE file(s) and then searches for
 this device in the AGGREGATE_DATA_FILE.
Allowed options:
Generic options:
  -h [ --help ]                         Produce help message

  -v [ --version ]                      Print version string

  -c [ --config ] arg (=config/disaggregate.conf)
                                        Configuration filename.
Configuration options:
  -s [ --signature ] arg    Device signature file (without path).  Use
                            multiple -s options to specify multiple signature
                            files.  e.g. -s FILE1 -s FILE2  Options specified
                            on command line will be combined with  options
                            from the config file.
  -n [ --device-name ] arg  The device name e.g. "kettle".
  -o [ --keep-overlapping ] Do not remove overlapping candidates during
                            disaggregation.
  --lms                     Use Least Mean Squares approach for matching
                            signature with aggregate data.
  --histogram               Use histogram approach. Full disaggregation is not
                            implemented for this approach hence an aggregate
                            file need not be supplied
  --cropfront arg           The number of samples to crop off the front of the
                            signature (only works with LMS or histogram).
  --cropback arg            The number of samples to crop off the back of the
                            signature (only works with LMS or histogram).
Example usage:
   ./disaggregate 10July.csv -s kettle.csv -s kettle2.csv -n kettle
```

A.1. Input files

Signature files are stored as `.csv` files in `data/input/watts_up`. These files have a single column recording the power consumption of the device sampled once a second. Several signature files are provided:

```
kettle.csv
kettle2.csv
toaster.csv
toaster2.csv
tumble.csv
washer.csv
washer2.csv
washer3.csv
washer4.csv
washer5.csv
```

Aggregate data files are stored as `.csv` files in `data/input/current_cost`. These files have two columns separated by a tab. The first column records a UNIX timestamp and the second column records the power consumption. Several aggregate data files are provided:

```
10July.csv
earlyJuly.csv
earlyAugust.csv
```

When calling ./disaggregate from the command line, do not prefix the file names with the directories; the directories are hard-coded.

A.2. Output files

All output files are sent to the `data/output` directory.

A.3. Configuration file

The command-line options for specifying signatures, aggregate data and device name can be supplied in the config file `config/disaggregate.conf`. For example:

```
device-name = washer
signature = washer.csv
signature = washer2.csv
aggdata = 10July.csv
```

The list of signature files specified in the config file and on the command line are merged into a single list.

A.4. GNUplot templates

The GNUplot template files live in the `config/` directory. See chapter 5.

A.5. Runtime dependencies

- `sed`
- `gnuplot`
- `graphviz`

A.6. Compiling from source

Compiling from source should be as simple as running `make` (the software was developed using gcc version 4.5.2).

A.6.1. Compilation dependencies

This project makes use of several Boost libraries. Boost version 1.42.0.1 was used during development. The specific libraries are:

Boost String Algorithms Library boost.org/doc/libs/1_42_0/doc/html/string_algo.html

Boost Graph Library boost.org/doc/libs/1_42_0/libs/graph/doc/index.html

Boost Unit Test Framework boost.org/doc/libs/1_42_0/libs/test/doc/html/index.html

Boost Program Options boost.org/doc/libs/1_42_0/doc/html/program_options - must be compiled from source and explicitly linked to. Details: boost.org/doc/libs/1_42_0/more/getting_started/unix-variants.html#easy-build-and-install

A.7. Generating Doxygen documentation

1. change directory to `doc`
2. run `doxygen` (you may have to install doxygen first! It is available from doxygen.org)
3. Now point a web browser at `doc/html/index.html`

A.8. Running unit tests

The full suite of tests can be run with `make testAll`. Alternatively, individual tests can be run with `make` <`testtarget`> where `testtarget` is one of the following: `ArrayTest, GNUplotTest, UtilsTest, StatisticTest, PowerStateGraphTest, AggregateDataTest`.

B. Further (simplified) code listings

B.1. Training code

Please note that this code listing has been simplified compared to the real code.

```
PSGraph::vertex_descriptor PowerStateGraph::updateOrInsertVertex(
        const Signature& sig,
        const Statistic<Sample_t>& postSpikePowerState )
{
  bool foundSimilar; // return param for mostSimilarVertex()
  PSGraph::vertex_descriptor vertex =
           mostSimilarVertex( &foundSimilar, postSpikePowerState );
  // mostSimilarVertex() uses a T-Test to find the powerStateGraph
  // vertex with a mean most similar to postSpikePowerState.

  if ( foundSimilar ) {
    if (vertex != offVertex)   // Update an existing vertex
      powerStateGraph[vertex].postSpike.update( postSpikePowerState );
  } else {
    // Add a new vertex
    vertex = add_vertex(powerStateGraph);
    powerStateGraph[vertex].postSpike = postSpikePowerState;
  }
  return vertex;
}

PSGraph::vertex_descriptor PowerStateGraph::mostSimilarVertex(
        bool * success, // return parameter.Did we find a satisfactory match?
        const Statistic<Sample_t>& stat, // stat to find in graph vertices
        const double ALPHA = 0.00000005  // significance level
) const {
  PSGraph::vertex_descriptor vertex=0;
  std::pair<PSG_vertex_iter, PSG_vertex_iter> vp;
  double tTest, highestTTest=0;

  // Find the best fit
  for (vp = vertices(powerStateGraph); vp.first != vp.second; ++vp.first) {
    tTest = stat.tTest( powerStateGraph[*vp.first].postSpike );
    if (tTest > highestTTest) {
      highestTTest = tTest;
      vertex = *vp.first;
    }
  }
  // Check whether the best fit is satisfactory
  *success = (highestTTest > (ALPHA/2) ); // T-Test
  return vertex;
}
```

```
void PowerStateGraph::updateOrInsertEdge(
        const PSGraph::vertex_descriptor& beforeVertex,
        const PSGraph::vertex_descriptor& afterVertex,
        const size_t samplesSinceLastSpike,
        const double spikeDelta )
{
  PSGraph::edge_descriptor existingEdge, newEdge;
  bool edgeExistsAlready; // return param from boost::edge()
  tie(existingEdge, edgeExistsAlready) =
          boost::edge(beforeVertex, afterVertex, powerStateGraph);

  if ( edgeExistsAlready ) {
    // update existing edge's stats
    powerStateGraph[existingEdge].delta.update( spikeDelta );
    powerStateGraph[existingEdge].duration.update( samplesSinceLastSpike );
  } else {
    // add a new edge
    tie(newEdge, edgeExistsAlready) =
          boost::add_edge(beforeVertex, afterVertex, powerStateGraph);
    powerStateGraph[newEdge].delta    = Statistic<double>( spikeDelta );
    powerStateGraph[newEdge].duration =
          Statistic<size_t>( samplesSinceLastSpike );
  }
}
```

B.2. Disaggregation code

Please note that this code listing has been simplified compared to the real code.

```
const PowerStateGraph::Fingerprint PowerStateGraph::initTraceToEnd(
        const AggregateData::FoundSpike& spike,
        const size_t deviceStart ) // The possible time the device started
{
    DisagTree disagTree;

    /* Omitted to save space:
        * add an "offVertex" to disagTree.
        * add a "firstVertex" representing first non-zero power state. */

    // add an edge between disagOffVertex and firstVertex
    add_edge( disagOffVertex,    // source vertex
              firstVertex,       // target vertex
              spike.likelihood, // edge value
              disagTree );

    // now recursively trace from this edge to the end
    traceToEnd( &disagTree, firstVertex, deviceStart );

    // find route through the tree with highest average edge likelihoods
    findListOfPathsThroughDisagTree( disagTree, disagOffVertex );

    // Return the most confident path through the disagTree
    return findBestPath( disagTree, deviceStart );
```

```
}

void PowerStateGraph::traceToEnd(
        DisagTree * disagTree_p,  // input and output parameter
        const DisagTree::vertex_descriptor& disagVertex,
        const size_t prevTimestamp // timestamp of previous vertex
) const {
  list<AggregateData::FoundSpike> foundSpikes;

  // A handy reference to make the code more readable
  DisagTree& disagTree = *disagTree_p;

  // base case
  if ( disagTree[disagVertex].psgVertex == offVertex )
    return;

  // For each out-edge from disagVertex.psgVertex, retrieve a list of
  // spikes which match and create a new DisagTree vertex for each match.
  PSG_out_edge_iter psg_out_i, psg_out_end;
  tie( psg_out_i, psg_out_end ) =
        out_edges( disagTree[disagVertex].psgVertex, powerStateGraph );

  for (; psg_out_i!=psg_out_end; psg_out_i++ ) {
    size_t begOfSearchWindow, endOfSearchWindow;
    const size_t WINDOW_FRAME = 8; // number of seconds to widen window by
    size_t e = powerStateGraph[*psg_out_i].duration.nonZeroStdev();

    begOfSearchWindow = (disagTree[disagVertex].timestamp +
    powerStateGraph[*psg_out_i].duration.min)- WINDOW_FRAME - e;

    endOfSearchWindow = (disagTree[disagVertex].timestamp +
    powerStateGraph[*psg_out_i].duration.max) + WINDOW_FRAME + e;

    //*****************************************************//
    // get a list of candidate spikes matching this PSG-out-edge //
    foundSpikes.clear();
    foundSpikes = aggData->findSpike(
                  powerStateGraph[*psg_out_i].delta,  // spike stats
                  begOfSearchWindow, endOfSearchWindow );

    //*******************************************************//
    // for each candidate spike, create a new vertex in disagTree //
    // and recursively trace this to the end                      //
    for (spike=foundSpikes.begin(); spike!=foundSpikes.end(); spike++) {

      double normalisedLikelihoodForTime =
          powerStateGraph[*psg_out_i].duration.normalisedLikelihood(
                spike->timestamp - disagTree[disagVertex].timestamp);

      // merge probability for time and for spike delta
      double avLikelihood =
            ( normalisedLikelihoodForTime + spike->likelihood ) / 2;
```

```
      // create new vertex
      DisagTree::vertex_descriptor newVertex=add_vertex( disagTree );

      // add details to newVertex
      disagTree[newVertex].timestamp = spike->timestamp;

      // get vertex that *psg_out_i points to
      disagTree[newVertex].psgVertex = target(*psg_out_i, powerStateGraph);

      // create new edge
      DisagTree::edge_descriptor newEdge;
      bool existingEdge;
      tie( newEdge, existingEdge ) =
              add_edge( disagVertex, newVertex, avLikelihood, disagTree );

      // recursively trace to end.
      traceToEnd(disagTree_p, newVertex, disagTree[disagVertex].timestamp);
    }
  }
}

void PowerStateGraph::findListOfPathsThroughDisagTree(
        const DisagTree& disagTree,
        const DisagTree::vertex_descriptor vertex,
        const LikelihoodAndVertex lav,
        list<PowerStateGraph::LikelihoodAndVertex> path
        // Deliberately called-by-value because we want a copy.
) {
  path.push_back( lav );

  // base case = we're at the end
  if ( vertex != 0 ) {
    listOfPaths.push_back( path );
    return;
  }

  // iterate through each out-edge
  Disag_out_edge_iter out_e_i, out_e_end;
  tie( out_e_i, out_e_end ) = out_edges( vertex, disagTree );

  for (; out_e_i!=out_e_end; out_e_i++) {
    downstreamVertex = target( *out_e_i, disagTree );
    LikelihoodAndVertex nextLav;
    nextLav.vertex = downstreamVertex;
    nextLav.likelihood = disagTree[*out_e_i];

    // we haven't hit the end yet so recursively follow tree downwards.
    findListOfPathsThroughDisagTree(
            disagTree, downstreamVertex, nextLav, path );
  }
  return;
}
```

C. Software engineering tools used

- Coding in C++ (using some C++0x features implemented in gcc 4.4.3 which is the gcc version install on the DoC machines)
- Code documentation: Doxygen / Javadoc
- Exploring and visualising data in GNUplot, MatLab and Open Office Calc (which does a better job of interpreting dates in imported CSV files than MatLab)
- IDE: Eclipse 3.6.2 with CDT 7.0.2
- Debugger: GDB (using Eclipse as the interface to GDB)
- make
- Unit testing framework: Boost.Test (with Eclipse Parser)
- Revision control: git (which also works as a simple backup system; the git repository on my laptop is cloned to my DoC folder)
- Graph visualisation: graphviz

Bibliography

[1] Current cost's 'Bridge' uses pachube for backend data management | pachube.community. [blog post] Retrieved 20 Aug 2011. URL: http://community.pachube.com/node/436.

[2] How the current cost bridge posts data. Current Cost Technical Blog. [blog post] Retrieved 20 Aug 2011. URL: http://currentcost.posterous.com/how-the-current-cost-bridge-posts-data.

[3] OFGEM. Renewables Obligation. [Web page] Retrieved 28 Aug 2011. URL: http://www.ofgem.gov.uk/Sustainability/Environment/RenewablObl/Pages/RenewablObl.aspx.

[4] Trends in carbon dioxide sampled from Mauna Loa, US National Oceanic & Atmospheric Administration Research website. Retrieved 24 Aug 2011. URL: http://www.esrl.noaa.gov/gmd/ccgg/trends/.

[5] Fourth assessment report: Climate change 2007 (AR4). Technical report, IPCC, 2007. URL: http://www.ipcc.ch/publications_and_data/publications_and_data_reports.shtml.

[6] UK climate change act, November 2008. UK Parliament. URL: http://www.legislation.gov.uk/ukpga/2008/27/contents.

[7] Synthesis report: Climate change, global risks, challenges & decisions. Technical report, International Alliance of Research Universities: Australian National University, ETH Zürich, National University of Singapore, Peking University, University of California - Berkeley, University of Cambridge, University of Copenhagen, University of Oxford, The University of Tokyo, Yale University, March 2009. URL: http://www.pik-potsdam.de/news/press-releases/files/synthesis-report-web.pdf.

[8] Centrica Plc, March 2010. British Gas Plans Two Million Smart Meters in British homes by 2012 [Press release]. URL: http://www.centrica.com/index.asp?pageid=39&newsid=1970.

[9] Smart meter implementation strategy prospectus. Technical report, DECC, Ofgem/Ofgem E-Serve, July 2010. URL: http://www.ofgem.gov.uk/e-serve/sm/Documentation/Documents1/Smart%20metering%20-%20Prospectus.pdf.

[10] White paper on smart grids - recommendations of the ICT industry for an accelerated SmartGrids 2020 deployment. Technical report, Digital Europe, June 2010. URL: http://www.digitaleurope.org/fileadmin/user_upload/document/White_Paper_on_Smart_1277195377.pdf.

[11] World energy outlook. Technical report, International Energy Agency, 2010. URL: http://www.iea.org/weo/docs/weo2010/WEO2010_ES_English.pdf.

[12] Arctic sea ice thickness. US Naval Oceanographic Office (NAVO) website, September 2011. URL: http://www7320.nrlssc.navy.mil/hycomARC/navo/arcticictnnowcast.gif.

[13] Climate Change Levy. HMRC website, 2011. URL: http://customs.hmrc.gov.uk/channelsPortalWebApp/channelsPortalWebApp.portal?_nfpb=true&_pageLabel=pageExcise_InfoGuides&propertyType=document&id=HMCE_CL_001174.

[14] Digest of UK energy statistics (DUKES) chapter 5: Electricity. Technical report, UK Department for Energy and Climate Change, 2011. URL: http://www.decc.gov.uk/assets/decc/11/stats/publications/dukes/2307-dukes-2011-chapter-5-electricity.pdf.

[15] Energy price statistics. Technical report, Department of Energy and Climate Change, webpage, 2011. URL: http://www.decc.gov.uk/en/content/cms/statistics/energy_stats/prices/.

[16] GISTEMP surface temperature analysis: Analysis graphs and plots, 2011. NASA Goddard Institute Of Space Studies website. URL: http://data.giss.nasa.gov/gistemp/graphs/.

[17] Inflation report. Technical report, Bank of England, London, August 2011. URL: www.bankofengland.co.uk/publications/inflationreport/ir11aug4.ppt.

[18] Polarstern reaches north pole. Arctic Sea Ice Blog, August 2011. URL: http://neven1.typepad.com/blog/2011/08/polarstern-reaches-north-pole.html.

[19] Shale gas. Technical report, British Geological Survey website, 2011. URL: http://www.bgs.ac.uk/research/energy/shaleGas.html.

[20] Statistical review of world energy. Technical report, BP, June 2011. URL: http://www.bp.com/sectionbodycopy.do?categoryId=7500&contentId=7068481.

[21] UK climate change sustainable development indicator: 2010 greenhouse gas emissions, provisional figures and 2009 greenhouse gas emissions, final figures by fuel type and end-user. Technical report, UK Department of Energy and Climate Change, March 2011. URL: http://www.decc.gov.uk/assets/decc/Statistics/climate_change/1515-statrelease-ghg-emissions-31032011.pdf.

[22] Unconventional gas and implications for the LNG market - FACTS global energy. In *Pacific Energy Summit*, Jakarta, Indonesia, February 2011. URL: http://www.nbr.org/downloads/pdfs/eta/PES_2011_Facts_Global_Energy.pdf.

[23] Dave Abrahams. Want speed? pass by value. - C++Next, August 2009. URL: http://cpp-next.com/archive/2009/08/want-speed-pass-by-value/.

[24] Arctic Climate Impact Assessment, Arctic Monitoring Programme, Assessment, Program for the Conservation of Arctic Flora Fauna, , and International Arctic Science Committee. *Arctic climate impact assessment*. Cambridge University Press, 2005.

[25] Ken Caldeira and Michael E. Wickett. Oceanography: Anthropogenic carbon and ocean pH. *Nature*, 425(6956):365, 2003. URL: http://dx.doi.org/10.1038/425365a, doi:10.1038/425365a.

[26] Anny Cazenave. How fast are the ice sheets melting? *Science (New York, N.Y.)*, 314(5803):1250–1252, November 2006. PMID: 17053111. doi:10.1126/science.1133325.

[27] John A. Church and Neil J. White. A 20th century acceleration in global sea-level rise. *Geophysical Research Letters*, 33:4 PP., January 2006. URL: http://www.agu.org/journals/ABS/2006/2005GL024826.shtml.

[28] W. Dansgaard, S. J. Johnsen, H. B. Clausen, D. Dahl-Jensen, N. S. Gundestrup, C. U. Hammer, C. S. Hvidberg, J. P. Steffensen, A. E. Sveinbjornsdottir, J. Jouzel, and G. Bond. Evidence for general instability of past climate from a 250-kyr ice-core record. *Nature*, 364(6434):218–220, July 1993. doi:10.1038/364218a0.

[29] S. Darby. The effectiveness of feedback on energy consumption. a review for DEFRA of the literature on metering, billing and direct displays. Technical report, Environmental Change Institute, University of Oxford, 2006. URL: http://www.eci.ox.ac.uk/research/energy/downloads/smart-metering-report.pdf.

[30] Andrew Duffy. Greens want smart meter plug pulled, July 2011. Times Colonist, Vancouver. URL: http://www.timescolonist.com/health/Greens+want+smart+meter+plug+pulled/5171603/story.html.

[31] Corinna Fischer. Feedback on household electricity consumption: a tool for saving energy? *Energy Efficiency*, 1(1):79–104, May 2008. doi:10.1007/s12053-008-9009-7.

[32] Jon Froehlich, Eric Larson, Sidhant Gupta, Gabe Cohn, Matthew Reynolds, and Shwetak Patel. Disaggregated End-Use energy sensing for the smart grid. *IEEE Pervasive Computing*, 10:28–39, January 2011. doi:10.1109/MPRV.2010.74.

[33] Aslak Grinsted, J. C. Moore, and S. Jevrejeva. Reconstructing sea level from paleo and projected temperatures 200 to 2100 ad. *Climate Dynamics*, 34(4):461–472, January 2009. doi:10.1007/s00382-008-0507-2.

[34] Christophe Guille and George Gross. A conceptual framework for the vehicle-to-grid (V2G) implementation. *Energy Policy*, 37(11):4379–4390, November 2009. URL: http://www.sciencedirect.com/science/article/pii/S0301421509003978.

[35] A.K. Gupta. Origin of agriculture and domestication of plants and animals linked to early holocene climate amelioration. *Current Science*, 87(1):54–59, 2004. URL: http://www.ias.ac.in/currsci/jul102004/54.pdf.

[36] James Hansen, Makiko Sato, Reto Ruedy, Ken Lo, David W. Lea, and Martin Medina-Elizade. Global temperature change. *Proceedings of the National Academy of Sciences*, 103(39):14288 –14293, 2006. doi:10.1073/pnas.0606291103.

[37] John E. Harries, Helen E. Brindley, Pretty J. Sagoo, and Richard J. Bantges. Increases in greenhouse forcing inferred from the outgoing longwave radiation spectra of the earth in 1970 and 1997. *Nature*, 410(6826):355–357, March 2001. doi:10.1038/35066553.

[38] G. W Hart. Residential energy monitoring and computerized surveillance via utility power flows. *IEEE Technology and Society Magazine*, 8(2):12–16, June 1989. doi:10.1109/44.31557.

[39] George W. Hart, Edward C. Kern, and Fred C. Schweppe. Non-intrusive appliance monitor, August 1989. United States Patent and Trademark Office. Patent number 4858141. URL: http://www.google.com/patents?vid=4858141.

[40] Ola M Johannessen. Decreasing arctic sea ice mirrors increasing CO2 on decadal time scale. *Atmospheric and Oceanic Science Letters*, 1(1):51–56, November 2008. URL: http://hdl.handle.net/1956/2840.

[41] Willett Kempton and Laura Montgomery. Folk quantification of energy. *Energy*, 7(10):817–827, October 1982. URL: http://www.sciencedirect.com/science/article/pii/0360544282900305, doi:16/0360-5442(82)90030-5.

[42] J. Z Kolter, S. Batra, and A. Y Ng. Energy disaggregation via discriminative sparse coding. *Neural Information Processing Systems (NIPS)*, 2010. URL: http://books.nips.cc/papers/files/nips23/NIPS2010_1272.pdf.

[43] C. Laughman, Kwangduk Lee, R. Cox, S. Shaw, S. Leeb, L. Norford, and P. Armstrong. Power signature analysis. *IEEE Power and Energy Magazine*, 1(2):56– 63, April 2003. doi:10.1109/MPAE.2003.1192027.

[44] J. Lenoir, J. C. Gégout, P. A. Marquet, P. de Ruffray, and H. Brisse. A significant upward shift in plant species optimum elevation during the 20th century. *Science*, 320(5884):1768 –1771, June 2008. URL: http://www.sciencemag.org/content/320/5884/1768.abstract, doi:10.1126/science.1156831.

[45] Dieter Luthi, Martine Le Floch, Bernhard Bereiter, Thomas Blunier, Jean-Marc Barnola, Urs Siegenthaler, Dominique Raynaud, Jean Jouzel, Hubertus Fischer, Kenji Kawamura, and Thomas F. Stocker. High-resolution carbon dioxide concentration record 650,000-800,000 years before present. *Nature*, 453(7193):379–382, May 2008. doi:10.1038/nature06949.

[46] Ian McDougall, Francis H. Brown, and John G. Fleagle. Stratigraphic placement and age of modern humans from kibish, ethiopia. *Nature*, 433(7027):733–736, February 2005. doi:10.1038/nature03258.

[47] Leslie K. Norford and Steven B. Leeb. Non-intrusive electrical load monitoring in commercial buildings based on steady-state and transient load-detection algorithms. *Energy and Buildings*, 24(1):51–64, 1996. URL: http://www.sciencedirect.com/science/article/pii/0378778895009582.

[48] James C. Orr, Victoria J. Fabry, Olivier Aumont, Laurent Bopp, Scott C. Doney, Richard A. Feely, Anand Gnanadesikan, Nicolas Gruber, Akio Ishida, Fortunat Joos, Robert M. Key, Keith Lindsay, Ernst Maier-Reimer, Richard Matear, Patrick Monfray, Anne Mouchet, Raymond G. Najjar, Gian-Kasper Plattner, Keith B. Rodgers, Christopher L. Sabine, Jorge L. Sarmiento, Reiner Schlitzer, Richard D. Slater, Ian J. Totterdell, Marie-France Weirig, Yasuhiro Yamanaka, and Andrew Yool. Anthropogenic ocean acidification over the twenty-first century and its impact on calcifying organisms. *Nature*, 437(7059):681–686, 2005. doi:10.1038/nature04095.

[49] Michael Parti and Cynthia Parti. The total and Appliance-Specific conditional demand for electricity in the household sector. *The Bell Journal of Economics*, 11(1):309–321, April 1980. ArticleType: research-article / Full publication date: Spring, 1980 / Copyright © 1980 The RAND Corporation. doi:10.2307/3003415.

[50] Paul N. Pearson and Martin R. Palmer. Atmospheric carbon dioxide concentrations over the past 60 million years. *Nature*, 406(6797):695–699, 2000. doi:10.1038/35021000.

[51] J. R. Petit, J. Jouzel, D. Raynaud, N. I. Barkov, J.-M. Barnola, I. Basile, M. Bender, J. Chappellaz, M. Davis, G. Delaygue, M. Delmotte, V. M. Kotlyakov, M. Legrand, V. Y. Lipenkov, C. Lorius, L. PEpin, C. Ritz, E. Saltzman, and M. Stievenard. Climate and atmospheric history of the past 420,000 years from the vostok ice core, antarctica. *Nature*, 399(6735):429–436, June 1999. doi:10.1038/20859.

[52] Leonid Polyak, Richard B. Alley, John T. Andrews, Julie Brigham-Grette, Thomas M. Cronin, Dennis A. Darby, Arthur S. Dyke, Joan J. Fitzpatrick, Svend Funder, Marika Holland, Anne E. Jennings, Gifford H. Miller, Matt O'Regan, James Savelle, Mark Serreze, Kristen St. John, James W.C. White, and Eric Wolff. History of sea ice in the arctic. *Quaternary Science Reviews*, 29(15-16):1757–1778, July 2010. URL: http://www.sciencedirect.com/science/article/pii/S0277379110000429.

[53] E. Rignot, I. Velicogna, M. R. van den Broeke, A. Monaghan, and J. Lenaerts. Acceleration of the contribution of the greenland and antarctic ice sheets to sea level rise. *Geophysical Research Letters*, 38:5 PP., March 2011. URL: http://www.agu.org/pubs/crossref/2011/2011GL046583.shtml.

[54] Johan Rockstrom, Will Steffen, Kevin Noone, Asa Persson, F. Stuart Chapin, Eric F. Lambin, Timothy M. Lenton, Marten Scheffer, Carl Folke, Hans Joachim Schellnhuber, Bjorn Nykvist, Cynthia A. de Wit, Terry Hughes, Sander van der Leeuw, Henning Rodhe, Sverker Sorlin, Peter K. Snyder, Robert Costanza, Uno Svedin, Malin Falkenmark, Louise Karlberg, Robert W. Corell, Victoria J. Fabry, James Hansen, Brian Walker, Diana Liverman, Katherine Richardson, Paul Crutzen, and Jonathan A. Foley. A safe operating space for humanity. *Nature*, 461(7263):472–475, 2009. `doi:10.1038/461472a`.

[55] K. von Schuckmann, F. Gaillard, and P.-Y. Le Traon. Global hydrographic variability patterns during 2003–2008. *Journal of Geophysical Research*, 114:17 PP., September 2009. URL: `http://www.agu.org/pubs/crossref/2009/2008JC005237.shtml`.

[56] Axel J Schweiger, Ron Lindsay, Jinlun Zhang, Michael Steele, Harry L. Stern, and Ron Kwok. Uncertainty in modeled arctic sea ice volume. *Journal of Geophysical Research*, 2011. `doi:10.1029/2011JC007084`.

[57] Dian J. Seidel, Qiang Fu, William J. Randel, and Thomas J. Reichler. Widening of the tropical belt in a changing climate. *Nature Geosci*, 1(1):21–24, January 2008. `doi:10.1038/ngeo.2007.38`.

[58] J. Seryak and K. Kissock. Occupancy and behavioral affects on residential energy use. In *American Solar Energy Society, Solar conference*, pages 717–722, Austin, Texas, 2003. URL: `http://www.sbse.org/awards/docs/2003/Seryak1.pdf`.

[59] S. R. Shaw, C. B. Abler, R. F. Lepard, D. Luo, S. B. Leeb, and L. K. Norford. Instrumentation for high performance nonintrusive electrical load monitoring. *Journal of Solar Energy Engineering*, 120(3):224–229, 1998. `doi:10.1115/1.2888073`.

[60] Robert H. Socolow. The twin rivers program on energy conservation in housing: Highlights and conclusions. *Energy and Buildings*, 1(3):207–242, April 1978. URL: `http://www.sciencedirect.com/science/article/pii/0378778878900038`.

[61] Susan Solomon, Gian-Kasper Plattner, Reto Knutti, and Pierre Friedlingstein. Irreversible climate change due to carbon dioxide emissions. *Proceedings of the National Academy of Sciences*, 106(6):1704 –1709, February 2009. `doi:10.1073/pnas.0812721106`.

[62] A. R. Stine, P. Huybers, and I. Y. Fung. Changes in the phase of the annual cycle of surface temperature. *Nature*, 457(7228):435–440, January 2009. `doi:10.1038/nature07675`.

[63] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison Wesley, 3 edition, June 1997.

[64] V Ismet Ugursal and Lukas G Swan. Modeling of end-use energy consumption in the residential sector: A review of modeling techniques. *Renewable and Sustainable Energy Reviews*, 13(8):1819–1835, 2009. `doi:10.1016/j.rser.2008.09.033`.

[65] Martin Vermeer and Stefan Rahmstorf. Global sea level linked to global temperature. *Proceedings of the National Academy of Sciences*, 106(51):21527 –21532, December 2009. `doi:10.1073/pnas.0907765106`.

[66] Muyin Wang and James E. Overland. A sea ice free summer arctic within 30 years? *Geophysical Research Letters*, 36(7), April 2009. `doi:10.1029/2009GL037820`.

[67] Richard A. Winett and Michael S. Neale. Psychological framework for energy conservation in buildings: Strategies, outcomes, directions. *Energy and Buildings*, 2(2):101–116, April 1979. URL: `http://www.sciencedirect.com/science/article/pii/0378778879900264`.

[68] R.W. Wood. Note on the theory of the greenhouse. *The London, Edinborough and Dublin Philosophical Magazine*, 17:319–320, 1909. URL: `http://www.wmconnolley.org.uk/sci/wood_rw.1909.html`.