Innovations in Non-Intrusive Load Monitoring

by

Andre Aboulian

B.S., Massachusetts Institute of Technology (2016)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2018

© Massachusetts Institute of Technology 2018. All rights reserved.
Innovations in Non-Intrusive Load Monitoring

by

Andre Aboulian

Submitted to the Department of Electrical Engineering and Computer Science on February 2, 2018, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

A non-intrusive load monitor (NILM) measures power at a central point in an electrical system to monitor the operation of many connected loads. Non-contact power meters sense the electromagnetic fields surrounding a cable and reconstruct the currents in each conductor. This thesis presents a novel calibration method for determining the relationship between fields and currents for a new meter by using an existing meter elsewhere within the distribution network.

A complex software pipeline captures, processes, and stores NILM data. Analysis tools are often not directly available to operators of mission-critical systems. An analytical and graphical platform is developed that provides diagnostic and prognostic information about equipment in real time to maintainers of electromechanical systems.

Thesis Supervisor: Steven B. Leeb
Title: Professor of Electrical Engineering and Computer Science

Thesis Co-Supervisor: John S. Donnal
Title: Assistant Professor of Weapons and Systems Engineering
Acknowledgements

Sponsors  This work has been made possible by the funding and support of the U.S. Coast Guard, the U.S. Naval Academy, the Office of Naval Research NEPTUNE Program, The Grainger Foundation, and the MIT-KUST program.

Colleagues  I would like to warmly acknowledge the assistance, advice, and encouragement of my colleagues: Daisy H. Green, Thomas J. Kane, Dr. Peter Lindahl, Gregory V. Bredariol, Spencer C. Shabshab, Eric A. Ponce, Manuel Gutierrez, Lukasz M. Huchel, Joshua Nation, Shibal Ibrahim, and Kevin W. Stevens.

Undergraduate Research Assistant  I owe a great deal of my success to the help of my undergraduate research assistant, Jennifer F. Switzer. She made significant contributions to the NILM Dashboard and NILM Console. I’m proud to have watched her develop new software skills and become more assertive in her work. And I am grateful to her for persisting even through my own doubts and shifting specification.

Friends  I want to thank my friends, who have supported and cared for me. I have built many bonds while at MIT, and I hope some of these relationships will last for a lifetime. To my roommates, I apologize for my odd hours and eccentricities.

Family  To my parents, whose hands are worn and eyes are tired from a lifetime of working. And who still lament from time to time for not being able to give me everything I wanted. You have made everything possible and worthwhile. Քաղաքացիական քաղաքացիական տար. ռուս.

\[1\] [Armenian] I am tremendously grateful.
Contents

List of Figures 8

List of Tables 12

1 Introduction 13
   1.1 Non-Contact Metering ........................................ 14
   1.2 Shipboard Equipment Monitoring ............................... 16

2 Indirect Calibration of Non-Contact Power Meters 17
   2.1 Introduction .................................................... 18
   2.2 Non-Contact Metering .......................................... 20
      2.2.1 Current Reconstruction ................................... 20
      2.2.2 Calibration ................................................ 22
      2.2.3 Phase Rotations .......................................... 25
      2.2.4 Preprocessing for Calibration ............................ 26
   2.3 Indirect Calibration Algorithm ............................... 28
      2.3.1 Load Transients .......................................... 29
      2.3.2 Eliminate Multi-Phase Loads ............................. 32
      2.3.3 Match Transients ....................................... 33
List of Figures

2-1 Non-contact meter hardware ........................................... 19
2-2 Fields around a multi-conductor cable cross-section ............. 21
2-3 Section of a power distribution network ............................. 24
2-4 Sine fit harmonic envelopes on example I/V data .................. 27
2-5 Data flow overview for the indirect calibration process .......... 28
2-6 Current and sensor PREP streams for an example dataset ....... 29
2-7 Transient stream of PREP data ......................................... 31
2-8 Elimination of multi-phase load transients ......................... 32
2-9 Matching load transients to form sensing pair .................... 34
2-10 Magnetic field response to a single current ....................... 36
2-11 Components of rotation angle for an isolated current .......... 37
2-12 Indirectly-calibrated result for an example dataset ............. 39
2-13 Standard deviation functions applied to unit step ................. 41
2-14 Intermediate signals of the transient detector ................... 42
2-15 Analysis windows surrounding a load transient ................. 43
2-16 Steps of the alignment process ..................................... 45
3-1 Annotated bench setup for indirect calibration .................... 53
List of Tables

3.1 Calibration accuracy for the wye-connected laboratory experiment . . 55
3.2 Calibration accuracy for delta-connected laboratory experiment . . 56
4.1 Monitored Loads in Engine Room . . . . . . . . . . . . . . . . . . . 82
4.2 Accuracy of classifying on-events . . . . . . . . . . . . . . . . . . . 89
Chapter 1

Introduction

Electrical signals can provide diagnostic and prognostic insight into the operation of electromechanical equipment, track human activity, and indicate overall system health. A Non-intrusive load monitor (NILM) measures power at a single point within an electrical network to monitor many loads within the system [1]. The technology has been successfully employed to monitor mission-critical equipment in industrial and shipboard settings [2,3]. Non-intrusive load monitoring is an inexpensive and labor-saving alternative to installing a distributed network of sensors on individual loads.

The NILM is often installed at a main power entry and measures aggregate power consumption. Signal processing is used to disaggregate the operation of individual loads within the system. Load detection algorithms are an active area of NILM research [4,5]. Electrical loads tend to have unique power characteristics when they are energized. The shape, magnitude, and power factor of the inrush waveform can help to identify the activity of individual equipment.

Modern NILM devices measure currents and voltages at high frequencies on the
order of kilohertz, which has been made possible by inexpensive electronics and sensing technologies. However, storing and interpreting this information is an ongoing problem. “Wattsworth” is a software suite specifically developed for NILM application [6, 7]. The platform includes a high speed database called NilmDB for storing time-series data [8]. It is capable of storing years of information and retrieving decimated copies of lengthy intervals to conserve bandwidth. The platform also contains tools for acquiring electrical measurements by interfacing with metering hardware. A signal processing pipeline converts the raw sensor readings to currents and voltages. Harmonic envelopes are computed for these sinusoidal signals using the “sinefit” algorithm, which extracts real and reactive power components at fundamental and higher-order frequencies [9]. Finally, a web-based application called “NILM Manager” can be used to view captured data and analyze it using custom python-based filters [10].

1.1 Non-Contact Metering

Conventional power meters – including mechanical and solid state variants – often require ohmic contact for voltage measurements and must surround individual conductors for current measurements [11]. The magnetic field around a wire is proportional to the current flowing through it, and electric field changes are proportional to the voltage. Phase conductors are bundled into cables for electrical distribution, and the cables usually need to be spliced to install a meter. This requires a skilled electrician, poses safety risks, and incurs significant costs. Additionally, critical system often cannot tolerate downtime caused by the installation.

Non-contact meters [12] leverage the magnetic and electric fields around a cable bundle to measure the currents and voltages of the individual conductors within.
The meter can be installed around the outer cable jacket by a non-technical user, with no special requirements for the placement of the meter’s sensors. The ease of the installation allows these meters to be retrofitted onto critical equipment and main power feeds without an interruption of service.

The non-contact meter measures local fields at the surface of the cable insulation that can be linearly related to the currents and voltages of the conductors. The combination of currents measured at each point depends on the spatial positioning of the magnetic field sensors. A linear transformation must be determined to recover the currents from magnetic field measurements. Additionally, the phase rotations relative to the measured electric fields are required to extract the ratio of real and reactive power components. These properties can be experimentally determined through a calibration process. One developed method involves introducing controlled loads in the electrical system that the non-contact meter can use to determine the current transformation and phase rotations [12, 13].

This thesis presents a novel method called “indirect calibration”, which uses an existing meter within a radial distribution network to calibration a new non-contact meter. The loads monitored by the new meter must produce currents that the reference meter can also measure. The non-contact meter can be installed downstream of a building meter to submeter a panel, or it can monitor the whole electrical system by measuring upstream of an existing submeter. Hardwired loads in the system can be switched to produce changes that are sensed at both meters. The calibration process detects these load transients and uses them to relate the currents in the conductors to the fields measured at the non-contact meter. The process of indirect calibration is described in chapter 2. The algorithm is applied to real systems in laboratory and shipboard settings, as presented in chapter 3.
1.2 Shipboard Equipment Monitoring

Modern ships rely on a series of critical systems for propulsion of the vessel and hospitality of the crew. Many pieces of equipment are controlled automatically through feedback and have no indicators of performance until they fail completely. Crew sizes on ships have decreased dramatically over the past several decades due to improved automation; however, much time is spent repetitively checking equipment for proper function \[14\]. Non-intrusive load monitors have been installed aboard several U.S. Coast Guard vessels to record electrical signals and detect faults \[15\]. Currently, this data is post-processed after each cruise and not used to diagnose active faults while the ship is underway. A Shipboard Automated Watchstander (SAW) \[16, 17\] could be developed to track the operation of critical equipment and indicate faults to the crew before they become catastrophic.

The “NILM Dashboard” described in chapter 4 is a software tool designed to provide diagnostic insights into the operation of shipboard equipment. The crew member on duty would be able to monitor many shipboard systems from a single user interface, only inspecting equipment when necessary. Electrical data is captured as time-series signals, which is too rich for presenting to the user directly. A load identification algorithm flags load transients and records them for later use. The dashboard can quickly retrieve these sparse events over extensive time ranges to display when each load was energized. This permits comparison of interdependent equipment and can be used to resolve discrepancies in human logs. Metrics about each load are shown live and also aggregated on a daily basis. For instance, these include the amount of power consumed, frequency of operation, and number of actuations. These statistics can indicate faults and assist with diagnosing failed components \[3\].
Chapter 2

Indirect Calibration of Non-Contact Power Meters

Non-contact power meters sense voltage and reconstruct the individual currents within a multi-conductor cable strictly from measurements of ambient electric and magnetic fields \[12,13\]. Non-contact meters offer comparable accuracy to conventional power meters, but with quicker installation and no need to make ohmic contact with a conductor. Typically, a known “calibration load” briefly provides data during installation so that non-contact meters can be calibrated to relate observed magnetic and electric fields to conductor currents and voltage. This paper demonstrates mathematical methods and experimental results for a new calibration technique that avoids the need for a calibration load in situations where other power meter data is available elsewhere on a radial distribution network.
2.1 Introduction

The electric power system can be used to monitor mission-critical electromechanical equipment for diagnostic and prognostic indicators of system health [2, 3, 15, 18]. Instrumenting individual equipment with sensors poses significant barriers in cost and labor. Instead, a non-intrusive load monitor (NILM) may be installed at a utility service entry or main electrical panel feed to monitor a collection of electrical loads. A NILM uses signal processing to disaggregate the operation of individual loads from the aggregate current data [1, 5]. The NILM has a “wide angle” or aggregate view of many loads, providing a relatively lower resolution for a given set of instrumentation electronics than would a dedicated submeter for a particular load. Signal processing can be used to enhance the dynamic range of a NILM so that relatively small changes in observed current can be reliably recognized [19].

Sometimes, however, it may be necessary to “zoom in” and monitor a reduced subset of loads on a branch of a radially distributed power network [20, 21]. Submetering a smaller section of a distribution network can enhance signal resolution and reduce the observed rate of transient event generation from the cycling of different loads on the system [4]. This enhanced secondary monitoring stream may be especially useful for diagnostic and prognostic fault detection applications [15].

Conventional power meters often require galvanic contact to voltage lines and must surround individual conductors with current transducers. Installation incurs downtime and significant cost [22]. Especially in retrofit installations on mission critical systems, the prospect of adding additional power system monitoring equipment may require an interruption of electrical service that cannot practically be tolerated.

Installing non-contact power meters [12] avoids the need for an interruption of electrical service. Non-contact meters can be simply secured to the outer insulation
of a multi-conductor cable, as shown in Fig. 2-1, with no special requirements for the placement of the sensors. This versatility permits installation in tight spaces and near critical equipment, where submetering might otherwise be difficult. And by keeping the installer safely on the outside of the insulation, non-contact meters obviate the need for skilled electricians.

This additional non-contact meter might be added to improve sensor resolution in amplitude and time while monitoring the operation of a subset of loads on the network. Non-contact meters must be calibrated upon installation to determine the mathematical relationship between observed fields and actual wire currents and voltage [12]. In situations where an existing power meter is already available somewhere on a radially distributed power distribution network, it is possible to use data from this existing power meter to calibrate a newly installed non-contact meter located elsewhere on the network. No special calibration load needs to be used to install the new meter. This paper presents a technique for calibrating non-contact meters using data from another power meter at a distinct location on the network. The ac-
accuracy of this method is demonstrated in a laboratory setting and also on shipboard microgrids as case studies.

2.2 Non-Contact Metering

The non-contact power meter described in [6,12,23] digitally samples magnetic and electric fields from several analog sensor nodes, which are zip-tied to the cable in Fig. 2-1. Each of the nodes has a hall effect sensor to measure the magnetic field and a capacitive pickup to measure changes in the electric field. Wires in the cable produce spatially-overlapping fields that are measured at the analog nodes, as depicted in Fig. 2-2. These measurements are linearly related to the voltages and currents of individual conductors. A series of analog-to-digital converters (ADC) sample the magnetic and electric field sensors, and a single microcontroller relays these samples to a host computer through a USB connection. Here, the sensors are sampled at 3kHz, providing significant resolution and bandwidth for use with 50Hz and 60Hz power systems. The data can be stored in a high-speed time-series database [8] and processed to reveal load operation for energy score-keeping and diagnostics [10].

2.2.1 Current Reconstruction

The magnetic fields generated by currents within a multi-conductor cable superimpose linearly, as modeled in [12]. Individual phase currents can be reconstructed from the magnetic field measurements by way of a linear transformation. Each current $I_n$ contributes to the sensed magnetic field $B_m$ by a factor of $K_{m,n}$. Equation (2.1) explicitly relates three independent currents to a vector of measurements from $M$ magnetic field sensors, for example. The matrix entries can be determined experi-
Figure 2-2: Cross-sectional view of a cable, whose conductors produce overlapping fields that linearly combine at the non-contact sensing nodes (gray). Each node \( m \) measures magnetic field \( (B_m) \) and electric field \( (E_m) \). The gradients around each conductor represent varying field strength.

mentally through a calibration process.

\[
\begin{bmatrix}
B_1 \\
B_2 \\
\vdots \\
B_M
\end{bmatrix} = \begin{bmatrix}
K_{1,1} & K_{1,2} & K_{1,3} \\
K_{2,1} & K_{2,2} & K_{2,3} \\
\vdots & \vdots & \vdots \\
K_{M,1} & K_{M,2} & K_{M,3}
\end{bmatrix} \begin{bmatrix}
I_A \\
I_B \\
I_C
\end{bmatrix}
\]

(2.1)

The calibration matrix \( K \) describes how the currents combine to produce magnetic fields. The inverse matrix, the current matrix \( K^+ \), specifies how the magnetic fields must be combined to recover the currents. Equation (2.2) shows the relationship
between these quantities.

\[ \vec{B} = \mathbf{K} \vec{I} \]
\[ \vec{I} = \mathbf{K}^+ \vec{B} \] (2.2)

The calibration matrix is invertible if there are at least as many linearly independent magnetic field sensors as there are independent currents. Since Kirchhoff’s current law (KCL) is obeyed at every point in the system, the number of independent currents \( N \) is one less than the number of non-ground conductors. In a wye-connected 3-phase (3\( \Omega \)) system, there are \( N = 3 \) independent line currents and 1 dependent neutral current. Therefore, there must be at least \( M \geq 3 \) sensing nodes around the power cable.

### 2.2.2 Calibration

A calibration process is any method that determines the \( \mathbf{K} \) matrix. This could be achieved by measuring the isolated effect of each current on each magnetic field sensor. For instance, a simple but impractical calibration strategy might be to pass a known constant current through only one conductor at a time. In terms of equation (2.1), the currents would be fixed at \( I_A = I_{cal} \) and \( I_B = I_C = 0 \), and the response of the magnetic field sensors \( \vec{B} \) would be measured during this time. The system of equations in (2.3) would then successfully determine the first column of \( \mathbf{K} \). However, this strategy is largely theoretical, since it is unrealistic to demand that some phase
currents are de-energized in a live system.

\[
B_1 = K_{1,1} \cdot I_{\text{cal}} \\
B_2 = K_{2,1} \cdot I_{\text{cal}} \\
\vdots \\
B_M = K_{M,1} \cdot I_{\text{cal}}
\]  

(2.3)

The calibration method discussed in \cite{12} also requires introducing a known load on each phase. However, this method operates on current changes rather than absolute levels, precluding the need to disable any operating equipment. A load transient occurs every time the calibration load turns on or off, causing a change in current level. Equation (2.2) can be rewritten as \( \partial \bar{B} = K \partial \bar{I} \). Similar to before, \( \partial I_A = I_{\text{cal}} \) and \( \partial I_B = \partial I_C = 0 \) while \( \partial \bar{B} \) is measured. Since \( \bar{I} \) and \( \bar{B} \) are linearly related, the \( K \) matrix can also be determined from changes in these quantities. This yields equation (2.4), which is similar in form to equation (2.2). The calibration load must be cycled at an identifiable frequency or with some known pattern, so that \( \partial \bar{B} \) can be found, for example, with an FFT. In this paper, this strategy is referred to as direct calibration.

\[
\partial \bar{B} = K \ \partial \bar{I}
\]  

(2.4)

Although direct calibration works well, coinciding transients from existing equipment can distort \( \partial \bar{B} \). Additionally, the calibration load must be sized and switched to achieve an observable, predictably-timed current waveform. A significant limitation may be the installer’s ability to energize the test load at all. The electrical network in Fig. 2-3 demonstrates the problem. Whereas the main panel has an avail-
Figure 2-3: Section of a power distribution network. The main panel is monitored with a calibrated meter, whereas the subpanel is instrumented with an uncalibrated non-contact meter.

With a common receptacle, the subpanel has no means for temporarily powering the calibration load. With direct calibration, there is no way for a non-contact meter to monitor this subpanel. This situation has occurred, for instance, in case studies aboard USCG Cutters [15, 18].

The *indirect calibration* process addresses the submeter calibration problem by exploiting the structure of the distribution network. All current delivered to the subpanel must first flow through the main panel. Since a calibrated meter already exists on the main panel in Fig. 2-3, this reference data can be leveraged to calibrate the submeter. The existing hardwired loads downstream of both meters are used instead of calibration loads. These existing loads create transients that are not precisely timed or sized. A *transient detector* (§2.4) enables detection of load events from current and magnetic field data. This paper will demonstrate the calibration
of submeters, but the indirect calibration method is also symmetrically capable of calibrating an upstream non-contact meter using an existing calibrated submeter.

2.2.3 Phase Rotations

Direct calibration methods can employ a load with a known power factor. For example, $I_n$ might assumed to be in phase with $V_n$ for a resistive calibration load. On the other hand, indirect calibration relies on in-situ loads that may have an unknown power factor. Real and reactive components of observed power will be known to the calibrated meter in the indirect scenario. However, the phase relationship between the sensed electric and magnetic fields for the as-yet uncalibrated downstream (in this scenario) meter are arbitrary, depending not only on the load but also on sensor placement.

This arbitrary phase relationship must be carefully resolved to correspond to the actual power factor angle between the currents and voltages in the cable. The calibration process begins by selecting the electric field sensor with the largest amplitude to act as the phase reference $E_{ref}$. The calibration process must determine the angle between $E_{ref}$ and each actual line voltage $V_n$. This quantity is referred to as the phase rotation $\phi_n$ and provides a way to track each line voltage using only $E_{ref}$.

The capacitive pickups sense the time derivative of the electric field, so these signals are digitally integrated upon measurement. Since the voltages are sine waves with identical frequency and similar amplitude, the combination of their fields at a particular point would be attenuated in amplitude and shifted in phase. The phase from the signal of a particular sensing node most closely resembles the nearest conductor’s voltage, so slight voltage imbalances relative to the other conductors do little to shift this reference. In scenarios where the power system voltage is
reasonably balanced, voltage amplitude is scaled proportional to $E_{\text{ref}}$ to arrive at per-phase power waveforms. Per-phase voltages can be reconstructed individually for more complex monitoring scenarios \cite{24}.

### 2.2.4 Preprocessing for Calibration

Sampling rates on the order of kilohertz are used to record currents and voltages at the calibrated meter, and also to record electric and magnetic fields from the sensors to be indirectly calibrated. Direct comparison of these AC waveforms to detect load transients is mathematically difficult and may be computationally expensive. To make the signals more manageable for the indirect calibration algorithm, they are converted to harmonic envelopes using the sine-fit algorithm \cite{9}. The sine-fit algorithm is used to extract the in-phase ($\mathcal{I}$) and quadrature ($\mathcal{Q}$) components of a signal at the fundamental frequency, as shown in Fig. 2-4. In this paper, a sine-fit harmonic envelope is called a preprocessed (PREP) stream and denoted as $Y$. This stream of complex numbers can also be expressed as a combination of its scalar components, as in equation (2.5). The real and imaginary parts correspond to in-phase and quadrature harmonic components, respectively. The magnitude $|Y|$ and angle $\angle Y$ can be used to describe the stream data in polar form.

$$Y = \mathcal{I}_Y + j \cdot \mathcal{Q}_Y$$  \hspace{1cm} (2.5)

The sine-fit algorithm uses a phase reference and set of rotation angles $\phi_n$ to compute in-phase and quadrature components. The phase reference is the electric field $E_{\text{ref}}$ for a non-contact meter, but it may also be a true voltage reference $V_{\text{ref}}$ for a conventional meter. The algorithm works by detecting the zero-crossings in the reference stream, which define the interval of each line cycle. The input signal
Figure 2-4: An example of Sinefit applied to a current signal, referenced to its voltage waveform. The in-phase ($I$) and quadrature ($Q$) components of the resulting PREP stream were scaled by the RMS voltage to depict power. Figure adapted from [9].

is analyzed over this window to determine its $I$ and $Q$ components, which are then rotated by the specified angle. Sinefit emits a single complex data point per line crossing. Therefore, the PREP stream is robust against different samples rates and clock skew present in the input signal. The next section describes how the indirect calibration process determines the correct rotation angles and calibration matrix for a new installation, given data from a calibrated reference meter elsewhere in the power distribution network.
2.3 Indirect Calibration Algorithm

A reference meter for indirect calibration provides accurate current and voltage ($I/V$) data, while the uncalibrated non-contact meter measures raw electric and magnetic fields ($E/B$) at a new installation point. The calibration process described below uses this data to determine the calibration matrix $K$ and the phase rotations $\{\phi_1..\phi_N\}$ for the uncalibrated meter.

Figure 2-5 shows an overview of the calibration algorithm. Sinefit converts data from the reference and uncalibrated meters to spectral envelopes to facilitate transient event detection and time alignment of observations between the two meters. Sinefit produces $N$ complex-valued PREP streams, one for every current of the reference meter. Each is referred to as a current stream ($C_n$). These provide an accurate view of the phase currents, since the current transformation and phase rotations are already known. For the uncalibrated meter, sinefit generates $M$ sensor streams ($S_m$). Each one describes the magnetic field at a particular sensing node. The strongest electric field $E_{ref}$ is used as a phase reference, and the rotations are initialized at $0^\circ$ since they have yet to be determined. The PREP streams in Fig. 2-6 correspond to
Figure 2-6: Current PREP streams $C_n$ (left) from the reference meter and sensor PREP streams $S_m$ (right) from an uncalibrated downstream non-contact submeter. An example dataset that will be used throughout this section.

The reference meter is sampled independently of the uncalibrated non-contact meter, causing a time offset between the two sources of data. The data is aligned to ensure that matching load transients occur at the same time instant in all streams. This process is described in §2.5.2. Once the PREP streams are aligned, the indirect calibration algorithm can proceed as detailed in Alg. 1.

### 2.3.1 Load Transients

The first step in the process involves identifying load transients in the calibrated meter data that correspond to magnetic field transients observed by the uncalibrated meter. A new stream that describes the changes in a scalar-valued signal $X$ is
Algorithm 1 The Indirect Calibration Process

**Input:** The reference meter provides currents \( \{I_1..I_N\} \) and voltage reference \( V_{\text{ref}} \), with phase rotations \( \{\gamma_1..\gamma_N\} \). The uncalibrated non-contact meter provides magnetic fields \( \{B_1..B_M\} \) and electric fields \( \{E_1..E_M\} \).

**Prepare Data Streams**

1: for each \( n \in \{1..N\} \) do
2: \( C_n \leftarrow \text{PREP}(I_n, V_{\text{ref}}, \gamma_n) \)
3: \( E_{\text{ref}} \leftarrow \text{MaxRMS}(\{E_1..E_M\}) \)
4: for each \( m \in \{1..M\} \) do
5: \( S_m \leftarrow \text{PREP}(B_m, E_{\text{ref}}, 0^\circ) \)
6: \( \text{ALIGN}(\{C_1..C_N\}, \{S_1..S_M\}) \)
7: for each \( n \in \{1..N\} \) do
8: \( \mathcal{D}(C_n) \leftarrow \text{DETECTTRANSIENTS}(C_n) \)
9: for each \( m \in \{1..M\} \) do
10: \( \mathcal{D}(S_m) \leftarrow \text{DETECTTRANSIENTS}(S_m) \)

**Calibration Algorithm**

11: \( \text{ELIMINATEMULTIPHASE}(\{\mathcal{D}(C_1)\ldots\mathcal{D}(C_N)\}) \)
12: for each \( n \in \{1..N\} \) do
13: for each \( m \in \{1..M\} \) do
14: \( \langle \mathcal{D}(C_{m,n}), \mathcal{D}(S_{m,n}) \rangle \leftarrow \text{MATCHTRANSIENTS}(\mathcal{D}(C_n), \mathcal{D}(S_m)) \)
15: for each \( n \in \{1..N\} \) do
16: \( i_{\text{mast}} \leftarrow \text{SELMASTER}(\{\mathcal{D}(S_{1,n})\ldots\mathcal{D}(S_{M,n})\}) \)
17: for each \( m \in \{1..M\} \) do
18: \( \text{scale} \leftarrow \text{CALCScale}(\mathcal{D}(C_{m,n}), \mathcal{D}(S_{m,n})) \)
19: \( \text{sign} \leftarrow \text{CALCSign}(\angle \mathcal{D}(S_{m,n}), \angle \mathcal{D}(S_{i_{\text{mast}},n})) \)
20: \( K_{m,n} \leftarrow \text{scale} \ast \text{sign} \)
21: \( \phi_n \leftarrow \text{CALCROT}(\angle \mathcal{D}(C_{i_{\text{mast}},n}), \angle \mathcal{D}(S_{i_{\text{mast}},n})) \)

**Output:** Calibration matrix \( K \) and phase rotations \( \{\phi_1..\phi_N\} \) for non-contact meter.
called the transient stream $\mathcal{D}(X)$. The process for generating the transient stream is described in §2.4. In this section, which overviews the entire calibration process, we assume that the transient stream $\mathcal{D}(X)$ contains non-zero points only at well-defined level changes.

A load transient causes a change in the $\mathcal{I}$ and/or $\mathcal{Q}$ components of a PREP stream $\mathcal{Y}$. The corresponding transient stream is also complex-valued and consists of the transient streams of the individual components, as expressed in equation (2.6). Transient streams $\mathcal{D}(\mathcal{C}_n)$ are determined for all currents and streams $\mathcal{D}(\mathcal{S}_m)$ are calculated for all magnetic field sensors. The relationship between a current PREP stream and its corresponding transient stream is illustrated in Fig. 2-7.

$$\mathcal{D}(\mathcal{Y}) = \mathcal{D}(\mathcal{I}_Y) + j \cdot \mathcal{D}(\mathcal{Q}_Y)$$ (2.6)
2.3.2 Eliminate Multi-Phase Loads

The calibration algorithm isolates the effects of each line current on the magnetic field sensors in order to determine the calibration matrix. In a wye-connected system, line-to-neutral in-situ loads provide a convenient way to isolate the effect of each line current on the field sensors. With these single-phase loads, all the return current must flow through the neutral conductor. Multi-phase loads, including line-to-line and $3\Omega$ loads, would require more mathematical and experimental manipulation for calibration, so they are typically excluded by focusing on data created by line-to-neutral loads. For indirect calibration to succeed, it is convenient to focus on single-phase load transients collected for each of the independent line-to-neutral currents. Other combinations are possible. For delta-connected systems, discussed in §3.2, the absence of a neutral connection makes line-to-line load transients most convenient for calibration.

It is necessary to eliminate all multi-phase load transients in the current data, so that they are not involved in the later calibration stages. The current transient
streams are scanned simultaneously to identify instances when transients occur on multiple currents at once. These points are removed from all current streams, effectively ignoring the multi-phase loads. This elimination is visually depicted in Fig. 2-8.

Mathematically, elimination is achieved by creating a mask $B_{\text{multi}}$, as specified in equation (2.7). This mask counts the load transients occurring across the current streams at each time instant. At each of the multi-phase transients, the mask contains a value greater than one. Each of the current transient streams $\mathcal{D}(C_n)$ is overwritten with zeros wherever $B_{\text{multi}} > 1$, effectively removing the multi-phase loads.

$$B_{\text{multi}} = \sum_{n=1}^{N} (1 \text{ if } |\mathcal{D}(C_n)| \neq 0, \text{ else } 0)$$ (2.7)

### 2.3.3 Match Transients

Each sensor measures the magnetic field resulting from a combination of line currents. A sensing pair is a set of streams that isolates the effect of each individual current on each sensor. The sensing pair consists of $\mathcal{D}(C_{m,n})$ and $\mathcal{D}(S_{m,n})$, which contain only the load transients from current $C_n$ detected by sensor $S_m$. There are $N \cdot M$ such combination overall. The pairs are formed from the transients shared among the corresponding current and sensor streams. These transients match when they occur at the same time instant across the two time-aligned streams. In practice, a binary mask (2.8) is used to isolate the transients occurring in the pair.

$$B_{\text{match}} = |\mathcal{D}(C_n)| \land |\mathcal{D}(S_m)|$$ (2.8)

When current transients occur in the absence of associated sensor transients, this means either that the sensor is located at a magnetic null for the current or
Figure 2-9: The current and sensor transient streams (left) combine to form a unique sensing pair (right). Dotted vertical lines indicate current transients. Annotated transients (red) do not pertain to the sensing pair.

that the current is flowing to branches upstream of an uncalibrated submeter. On the other hand, when sensor transients occur without matching current transients, the sensor is measuring a different phase current or an uncalibrated main meter is detecting branches upstream of the downstream reference meter. The sensing pair shown in Fig. 2-9 excludes upstream current transients (circles) and sensor transients from other phases (rectangles). Notice that the sensing pairs structurally mirror the calibration matrix, since each pair represents a sensor’s response to a particular current.

2.3.4 Determine Sensor Scales

The conductors’ magnetic fields combine linearly at each sensor. The responses of the sensors to the currents, in ADC counts per ampere, form the elements of the calibration matrix. Each element corresponds to a particular sensing pair. The associated transient streams are related by this ratio. Therefore, the next stage of
the calibration algorithm focuses on determining an unsigned sensor scale, which describes the magnitude of the response to a particular current.

The load transients can be conceptualized as points on a 2D plane, with current along the horizontal axis and magnetic field along the vertical. Corresponding transients from $|D(C_{m,n})|$ and $|D(S_{m,n})|$ pair to form points on this plane. The slope of the line passing through these points represents the sensor’s scale, and the line’s y-intercept is fixed at the origin to ensure linearity. Since the transient magnitudes are used, the slope is necessarily positive.

A least squares regression is applied in a two-step process to determine each sensor scale. A line is initially fitted to all the available transients points. Even in a perfect scenario, there may be outliers to this fit line. Namely, if an upstream load is energized at the same instant as a downstream one, a non-contact submeter would only detect the downstream change. This transient would incorrectly signify a lower sensor scale. Therefore, the points deviating significantly from the initial fit line are eliminated. The linear regression is performed with this reduced set for a tighter fit along the remaining points.

2.3.5 Determine Sensor Signs

The sensors do not necessarily measure an increase in magnetic field due to additional current. Depending on spatial orientation, a particular sensor may respond positively to one current and negatively to another. The example in Fig. 2-10 depicts the two possible responses. Although a single current is shown for simplicity, this is conceptually identical to single-phase load transients superimposed on a fully-energized multiphase system. This stage of the calibration algorithm determines the sign of each sensor response.
Figure 2-10: The response of two magnetic field sensors to a single current.

One magnetic field sensor is designated to be the *master sensor* for each current phase, arbitrarily defining a positively-signed sensor response to that phase. The sensor exhibiting the largest average value of $|\mathcal{D}(S_{m,n})|$ for a particular phase $n$ is selected to act as the master. The remaining sensing pairs for that phase are sequentially compared to that of the master sensor. If the angles of the sensor transients match, then that sensor’s sign is also positive. If the angles complement one another, the sensor coefficient is negatively-signed. Figure 2-10 illustrates how these angles indicate opposing signs. To set the value for a matrix element $K_{m,n}$, the sensor scale determined in the previous step is multiplied by its corresponding sign, as indicated in line 20 of Alg. 1.

### 2.3.6 Determine Rotations

The calibration matrix can reconstruct currents without explicitly considering the voltages on those conductors. However, the phase rotations for each current must be determined so that real and reactive power can be monitored. The rotations express how each phase voltage can be estimated by rotating $E_{ref}$ by various amounts.
Figure 2-11: The isolated effect of one current upon a magnetic field sensor. The phase offsets among the raw signals provide physical significance to the angles of the PREP streams.

The known currents have the correct $I/Q$ rotations on each phase, meaning that the angle at each current transient results solely from the power factor of that load ($\theta_{PF}$). The sensor PREP streams, however, are statically rotated to $0^\circ$. Each sensor transient angle results from the power factor, the known meter’s phase rotation ($\theta_{ph}$), the offset between voltage references ($\theta_{off}$), and a possible sign change ($\theta_{sign}$). The components of the current and sensor transient angles are shown in equations (2.9) and (2.10), respectively. An example relating these quantities with a set of raw signals.
appears in Fig. 2-11. For the sake of physical intuition, these waveforms depict the influence of one current upon a single magnetic field sensor. However, since the effects of multiple currents and loads are additive, the changes characterized by the load transient streams can be accurately represented as these isolated signals.

\[ \angle \mathcal{D}(C_n) = \theta_{PF} \]  

\[ \angle \mathcal{D}(S_m) = \theta_{off} + \theta_{ph} + \theta_{sign} + \theta_{PF} \]  

(2.10)

The load’s power factor angle \( \theta_{PF} \) influences the angle of each sensor transient as an additional correction over the phase rotation. Therefore, the desired rotation \( \phi_n \) for each phase is the difference between the sensor and current transient angles. Since the matrix elements were determined relative to the master sensor for each phase, the rotations are also calculated with respect to the corresponding master sensing pairs. Equation (2.11) computes the phase rotation at each transient. The index \( i_{\text{mast}} \) refers to the master sensor for phase \( n \). Although the rotation must be consistent across transients, a circular median filter helps to resolve the slight angular discrepancy among various loads, which is usually less than 1°.

\[ \phi_n = \angle \mathcal{D}(S_{i_{\text{mast}},n}) - \angle \mathcal{D}(C_{i_{\text{mast}},n}) \]

\[ = \theta_{off} + \theta_{ph} + \theta_{sign} \]  

(2.11)

This strategy for determining the phase rotations resolves the sign ambiguity of the master sensor. This sensor was previously assumed to respond positively to the phase’s current. If this assumption was incorrect, the reconstructed current for
Figure 2-12: Indirectly-calibrated result for an example dataset. Reconstructed currents from the subpanel (right), compared with the known currents from the main panel (left).

This phase would have the correct value but the opposite sign. In this situation, $\phi_n$ would be corrected with $\theta_{\text{sign}} = 180^\circ$ at every load transient, which reverses the sign automatically.

The calibration matrix and rotations are fully determined after the indirect calibration algorithm has completed. The calibration results are applied to the example dataset referenced throughout this section. The reconstructed currents appear in Fig. 2-12.

### 2.4 Load Transient Detector

The calibration process hinges on the ability to reliably detect load transients from experimental measurements. This section describes the transient detector function.
\( \mathcal{D} \), which produces a stream containing a non-zero point at each load transient. These points are located at sudden changes, such as the inrush waveform. The value corresponds to the amount of change in the steady-state level due to the transient.

Typical edge detection strategies are unsuitable for indirect calibration. A simple iterative approach may track the signal until it changes levels by a certain threshold \( \text{[1]} \). The threshold needs to be delicately tuned to overcome fluctuations while still detecting small changes, making this method highly dependent on the signal’s environment. Frequency-based filtering or convolving edge profiles alleviates the threshold sensitivity, but these filters are selective to transient shapes. A second-derivative Gaussian kernel provides versatile detection and offers decent locality \( \text{[25, 26]} \). However, indirect calibration cannot tolerate the time shifts introduced by these methods at each transient.

The transient detector \( \mathcal{D} \) uses a windowed standard deviation (STD). The trailing function \( \sigma_+ \) outputs the STD of the preceding \( W_T \) samples, while the leading function \( \sigma_- \) targets the next \( W_L \) samples. When either of these functions moves over a significant change, the output gradually rises then falls. Figure \( \text{2-13} \) shows these functions applied to the unit step \( H \), which is an ideal transient. These functions are largely useful because they return approximately zero over regions of constant-centered fluctuations. The volatility function in equation \( \text{(2.12)} \) quantifies the difference in deviation before and after each point. Since \( H \) is steady on either side of the step change, the \( \sigma_V(H) \) stream inflects at this point.

\[
\sigma_V = \sigma_+ - \sigma_-
\]  

(2.12)

The transient detector calculates the volatility stream of the input data \( Y \), which has been smoothed with a small median filter. The \( \sigma_V(Y) \) stream takes on the ap-
Figure 2-13: Various standard deviation functions applied to the unit step.

The appearance of $\sigma_V(H)$ near each transient. Cross-correlating these two volatility streams produces $R = corr(\sigma_V(H), \sigma_V(Y))$, which peaks at each transient, as shown in Fig. 2-14. The relative extrema of $R$ above an empirically-set threshold $\tau_{corr}$ become candidates locations for load transients. This threshold is specific to the type of input data (ex. currents) and establishes the minimum change that would be considered a valid transient. Since the STD function attenuates small fluctuations in the input signal, $\tau_{corr}$ does not vary greatly between environments.

The candidate transients must be verified against the input stream to ensure they correspond to acceptable changes. If a transient occurs within $W_T$ before or within $W_L$ after another transient, they are both discarded. Then, the samples on either side of the remaining transients are ignored, since they are likely to vary dramatically. The farther samples are used to ascertain steady-state properties. These regions are visually depicted in Fig. 2-15 relative to the window sizes used for the STD functions in previous steps. Notice these regions are asymmetrical, since the startup waveform may be unsteady for many line cycles.

The STD is computed over the valid regions before and after the transient, yield-
Figure 2-14: Intermediate signals of the transient detector. Dotted vertical lines represent candidate transient locations.

\( \sigma_B \) and \( \sigma_A \) respectively. The difference \( \sigma_{\text{bal}} = \text{abs}(\sigma_A - \sigma_B) \) must be below a small threshold \( \tau_{\text{bal}} \) to signify that the fluctuations are “balanced” on either side. If this check passes, then averages \( Y_A \) and \( Y_B \) are taken over the same regions. Their difference quantifies the stead-state change uses in the output stream \( D(Y) \), which appears in Fig. 2-14. Notice that some candidate transients do not appear in this stream since they failed various verification checks.

Transient detection may be performed on one scalar stream at a time. For complex PREP streams, it is preferable to locate transient candidates on \( |X| \) rather than on the dependent \( I_X \) and \( Q_X \) components individually. This way, detection is not influenced by rotation. Verifying that \( \sigma_{\text{bal}} < \tau_{\text{bal}} \) occurs on the components individually, although the candidate is discarded if the check fails for either. Finally, the steady-state change is computed for each component. This special handling of PREP streams ensures that when a transient appears on \( D(I_X) \), it must also appear at the same location on \( D(Q_X) \).
2.5 Sample Alignment

The calibrated and uncalibrated meters are often separated by significant physical distance in an electrical network. The measurements are obtained independently of one another, experiencing different timing offsets and skew. Once the data is collected, information must be synchronized between the two sources.

2.5.1 Synchronous Sampling

Timing skew can sometimes pose issues in high-frequency data. Clock instability on the meters’ embedded electronics may produce a different number of samples than expected at the programmed rate. The computer recording these samples may also experience time drift on its system clock. These effects combine to produce unreliable timestamps in absolute terms, meaning that two asynchronous streams would be mismatched relative to another.

Surprisingly, timing skew can be ignored altogether for a pair of continuous datasets. Reliable timing information is available in the data itself, since the electrical
line cycles can be utilized as a global clock. Sinefit produces exactly one data point per line cycle, ensuring that each sample in the current PREP stream corresponds to a single point in the sensor RPEP stream. Consequently, frequency variations over time are not a concern since the number of samples is conserved.

2.5.2 Time Offset Compensation

The calibrated and uncalibrated meters are often separated by significant distance within a facility, requiring a separate NILM computer for each. The system clocks on these acquisition computers often have a significant time offset. Internet time synchronization via the dominant Network Time Protocol (NTP) service typically leads to about 100 milliseconds of error. Even if these timestamps were derived from an ideal reference, the first timestamp will be unlikely to exactly correspond to the first ADC sample due to the communication delays inherent in the meters’ asynchronous protocols of USB and Ethernet. In practice, the system clocks in most offline environments are set manually within a few seconds of acceptable error. Since misalignment is an insurmountable aspect of acquisition, the data must be synchronized prior to calibration.

The PREP data samples are in lockstep across multiple sources in the same network. The streams, however, are offset from one another by an unknown number of samples. The possible offset is estimated to be within \( t = 60 \) seconds for the \( r = 60 \) hertz PREP stream. PREP data points carry timestamps that correspond to those occurring midway through the line cycle on the raw source. Although the timestamps are susceptible to drift over the acquisition interval, the effect is negligible in the low-frequency PREP data and amounts to a few samples of inaccuracy over the course of an hour.

44
Figure 2-16: Steps of the alignment process. Data streams (gray) are summed from the diffs of individual currents and sensors.

The current and sensor PREP streams may cover varying lengths of time. The initial step is to isolate the interval where they overlap. The timestamps provide an acceptable means for roughly determining the start of the region, which occurs at the earliest timestamp common to both streams \( t_b \). If available, an additional \( r \times t \) data points are retaining prior to \( t_b \) to avoid truncating useful information. If there are fewer than \( r \times t \) points, the remaining spaces are filled with the first value in the stream. The end of the overlapping region is not determined by timestamp. Instead, the stream with the fewest samples after \( t_b \) determines whether the other stream will be truncated. The end is padded by \( r \times t \) points with a similar method to the beginning. The resulting streams are exactly equal in length and roughly aligned in time.

The streams are ready to be finely synchronized. However, all that is known about the current and sensor streams prior to calibration is their changes should occur at the same time. It is unknown which sensor relates to which current. There-
fore, the current streams are summed together as \( \sum_{n=1}^{N} \Delta C_n \) to produce a single aggregate stream which includes all the changes. Similarly, the sensor streams are combined into \( \sum_{m=1}^{M} \Delta S_m \). The diff function (\( \Delta \)) computes the value difference between successive data points, like a discrete-time derivative. The diff function is preferable to the transient detector for alignment since it captures the shape of load currents and does not introduce independent time shifts at each transient. To use the aggregate streams, one is held stationary while the other is progressively shifted until the streams align. A standard cross-correlation function is used to determine the time shift which yields the greatest match between streams. Finally, the current and sensor streams are permanently shifted to correct this time offset.

2.5.3 Transient Offset Tolerance

The calibration algorithm requires a particular load transient to occur at the same instant across all current and sensor transient streams. A load can turn on abruptly or cause an inrush current lasting several line cycles. The transient detector will produce a load transient at some point along this inrush waveform. However, the detector may position the same transient occurring on a different current phase or sensor stream at a slightly different time instant. This inconsistency is due to physical differences in these signals, such as scale and the effect of external loads, as well as parameters used in the transient detector. The transients may also be offset between currents and sensors due to imperfect alignment of their respective PREP streams, often by no more than a few samples. Despite the slight time offset, these points must be associated to the same load transient.

At various stages of the algorithm, it becomes necessary to permit a few samples of misalignment tolerance. In other words, two transients on different streams are
said to occur at the same time if they are no more than $\delta$ samples apart. The transient
detector guarantees that the minimum distance between transients is no less than
the size of its detection window. Therefore, the chosen value of $\delta$ must be smaller
than half of the window size for the transient detector, so as not to incorrectly match
with nearby transients. Offset tolerance is used to match transient between currents
to eliminate multi-phase loads in §2.3.2. It is also used to match transients between
current and transients streams in §2.3.3. In successive stages of the algorithm, the
transients have already been matched into sensing pairs and no longer require this
tolerance.

\section*{2.6 Delta Systems}

Delta-connected electrical systems appear in many industrial settings and are com-
mon in shipboard applications. Although the calibration process has been described
in the context of a wye-connected electrical system, the same methodology can be
applied as-is to a delta-connected electrical network. However, additional stages are
necessary before and after the existing process for this adaptation.

\subsection*{2.6.1 Interpreting as a Special Case}

Indirect calibration can be adapted a delta-connected system without any changes
to the fundamental steps of the algorithm. Kirchoff’s current law on a delta network
implies that one of the three phase currents is a dependent variable, carrying the
algebraic sum of the other two phase currents. Calibration is conducted with two
sequential applications of a line-to-line load. For instance, a load is first energized
between phases A and B; then, between C and B. In this case, phase B is analogous
to the neutral current in a wye system and will not be involved during the calibration process. Therefore, calibrating a delta-connected system requires at least one line-to-line transient on each of the two independent phases. As with the wye-connected network, any other multi-phase load that creates transients involving combinations of the independent phases is eliminated. By observing the fields created by the line-to-line calibration load applied on independent phases during each of these two serial tests, the algorithm can determine the current transformation and rotations of the independent phases.

2.6.2 Designating the Dependent Phase

An appropriate dependent phase cannot be selected at random. For instance, if there were only A-B and B-C loads available, phase B would need to be designated as the “neutral”. The remaining A and C current streams would appear to have line-to-neutral load transients on those phases. However, if A was the dependent phase, only B load transients would remain and the B-C loads would be eliminated since they are multi-phase. An additional constraint in selecting the dependent phase is that the sensors must pickup the independent phases. If all line-to-line load combinations were available but only conductors A and B were coupled to magnetic field sensors, then phase C must serve as the dependent phase.

The dependent phase is selected automatically by applying similar reasoning algorithmically. Each phase is assessed for its suitability as a “neutral” by looking at the usable transients on the remaining phases. The candidate dependent phase is denoted as $C_{cn} \mid cn \in \{1...N\}$ and $C_p \mid p \neq cn$ represents one of the remaining phases for a particular choice of phase. In this situation, a usable transient on phase $C_p$ is one which appears on that phase, the candidate dependent phase, and any of
the sensors. The variable $n_p$ reflects the number of such transients on phase $p$, as shown in \(2.13\).

$$n_p = \text{count} \left( \left| \mathcal{D}(C_p) \right| \land \left| \mathcal{D}(C_{cn}) \right| \land \sum_{m=1}^{M} \left| \mathcal{D}(S_m) \right| \right)$$

The minimum $n_p$ among the remaining phases becomes the rank $r_{c,n}$ of the candidate dependent phase. The rank of a phase expresses the favorability of that phase to act as the dependent phase, since it guarantees that each of the other phases has at least $r$ load transients. Naturally, the phase with the highest rank is selected as the dependent phase. In the examples above, the ranks would be $\vec{r} = (0 \ L \ 0)$ and $\vec{r} = (0 \ 0 \ L)$ respectively for $L$ load transients. Notice that the results of this algorithm match the intuitively assignment dependent phases in the examples.

\subsection{2.6.3 Recovering Matrix & Rotations}

The calibration process will produce a calibration matrix and a set of rotations for only the independent currents, since the dependent current stream was discarded beforehand. Recovering the dependent current is performed on the current matrix, which is the pseudoinverse of the determined calibration matrix. The current matrix describes the combination of magnetic fields needed to reconstruct each current. Since the dependent phase provided the return path for the calibration loads, its current can be derived as the sum of the independent currents. Therefore, the missing row of the current matrix is computed as the sum of the other rows and negated to correct for the opposite direction of the dependent current.

Recovering the rotation of the dependent phase is less trivial. The methods in [13] were adapted toward determining the final rotation, and the original paper should be consulted for a more thorough explanation. Recall that the rotations may
be either 120° or 60° apart in a 3∅ system. If the angle between the determined rotations is acute, one of the rotations must be flipped by 180°, effectively forcing a 120° separation between the phases. The corresponding row of the current matrix is negated in order to preserve the correct direction. The difference between these rotations now reveals which phase leads in the time domain. The final rotation is simply 120° added to the rotation of the lagging phase.

2.6.4 Implementation & Usage

The calibration pipeline has been implemented as a multifunctional python utility, the code for which appears in appendix[C.1]. The tool is able to acquire measurements from meter hardware and indirectly calibrate a non-contact meter using this data. A step-by-step guide for performing this calibration process in the field has been documented in appendix[A.1].

Since the calibrated and uncalibrated non-contact meters are often not co-located, the utility is used on separate computers to independently acquire measurements from these meters. During this time, suitable loads in the electrical network are switched to produce load transients. At the uncalibrated meter, the program captures raw magnetic ($B_m$) and electric ($E_m$) fields to a file. Using the same tool at the calibrated meter, the raw samples are captured and immediately converted to $I/V$ data and PREP streams ($C_n$). This is possible since the current transformation and phase rotations are known at this meter. The utility also provides a graphical means to view the captured signals before proceeding. The current streams from the reference meter are copied to the uncalibrated meter’s computer so that they can be used for calibration.

The calibration process is invoked with a command and proceeds automatically.
Optionally, intermediate stages can be plotted for debugging and parameters can be tuned for the transient detector. The NILM system is automatically configured with the conversion matrix, rotations, and meter information once the calibration succeeds. The non-contact meter is considered calibrated at this point. The NILM system will correctly reconstruct currents and determine harmonic envelopes the next time its capturing pipeline runs.
Chapter 3

Experimental Results of Indirect Calibration

The indirect calibration algorithm has been tested extensively in a laboratory setting. It has also been tested in field installations on the microgrids of U.S. Navy and U.S. Coast Guard vessels.

3.1 Lab Experiments

The indirect calibration method was tested in a laboratory setting under various metering configurations and with a diverse set of loads on a three-phase 120/208 volt five-wire service. The resulting calibrations were assessed on the basis of accurate current reconstruction to evaluate the end-to-end success of algorithm and sensor hardware described in this paper. The electrical topology of the testbed is similar to the simplified distribution network shown in Fig. 2-3. Non-contact meters monitor the main panel and the subpanel. These have been pre-calibrated through direct
A conventional power meter (C) monitors the main panel. Non-contact meters are installed on the multi-conductor cables leading to the subpanel (D) and main panel (E). A commodity computer (F) runs the calibration utility and the NILM software infrastructure.

calibration, allowing them to act as either reference meters or uncalibrated sensors.

The main panel was also simultaneously monitored by a conventional meter to provide cross-validation. This meter uses LEM LF305-S and LEM LV25-P transducers to measure per-phase currents and voltages, respectively. These analog measurements are digitally sampled with a LabJack UE9 DAQ. In this setup, controlled loads can be introduced on either the main panel or the subpanel. Loads on the subpanel appear on all meters, whereas main panel loads do not appear on the subpanel meters. The complete experimental setup is shown in Fig. 3-1.

Various types of loads are introduced on each panel during the experiment shown in Fig. 3-2. Incandescent bulbs and high-power resistors are connected line-to-neutral as real-power loads. These are cycled on each phase of the subpanel, while similar loads are energized on the main panel \( t = [0, 80] \). Some transients occur on different
Figure 3-2: Per-phase currents for the wye-connected laboratory experiment, including the main panel (left) and subpanel (right).

phases, while others deliberately coincide on the same phase. Similarly, real-power loads are cycled on the main panel while others are energized on the subpanel \((t = [80, 180])\). A lamp dimmer is connected to various phases of both panels, since it exhibits a significant non-unity power factor \((t = [180, 240])\). Finally, some \(3\phi\) and line-to-line loads are powered \((t = [240, 300])\).

Indirect calibration was performed using all possible combinations of meters. For instance, the pre-calibrated non-contact meter on the main panel is used to calibrate the subpanel non-contact meter, and vice versa. The results were checked for the relative accuracy of the meter calibrated with the proposed algorithm versus the pre-calibrated meter chosen as the reference (“known”) meter for each experiment.

The transients appearing on the reconstructed currents of the indirectly-calibrated meter are compared to those from the reference meter. Accuracy statistics for transient magnitude and power factor across all loads are presented in table 3.1.
Table 3.1: Calibration accuracy for the wye-connected laboratory experiment.

<table>
<thead>
<tr>
<th>Meters</th>
<th>Mag. Err. [%]</th>
<th>P.F. Err. [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Known</td>
<td>Uncalibrated</td>
</tr>
<tr>
<td>Main Conv.</td>
<td>Main N.C.</td>
<td>0.01</td>
</tr>
<tr>
<td>Main Conv.</td>
<td>Sub N.C.</td>
<td>0.02</td>
</tr>
<tr>
<td>Main N.C.</td>
<td>Sub N.C.</td>
<td>0.00</td>
</tr>
<tr>
<td>Sub N.C.</td>
<td>Main N.C.</td>
<td>0.01</td>
</tr>
<tr>
<td>Sub N.C.</td>
<td>Sub N.C.</td>
<td>0.01</td>
</tr>
<tr>
<td>Main N.C.</td>
<td>Main N.C.</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Figure 3-3: Error between the reference conventional meter and the indirectly-calibrated non-contact meter, both monitoring the wye-connected main panel.

A second method of error analysis appears in Fig. 3-3. The error vector magnitude (EVM) metric is plotted for all reconstructed transients. It is visually discernible that each phase aligns at a unique error level. This indicates that the error, albeit minimal, is due to the calibration rather than volatility in the sensor. In fact, this repeatability is present across different types of loads and despite other energized phases.

The setup is reconfigured for delta-connected measurements, and a similar experiment produces the currents in Fig. 3-4. Resistors and incandescent bulbs are connected line-to-line as single-phase loads. This type of connection naturally produces a phase rotation for real-power loads, which manifests as opposing \( Xim \) values on the involved line currents. Similar to before, these loads are cycled on the subpanel.
while others are energized on the main panel \((t = [0, 105])\). Later, loads are cycled on the main panel while others are held on the subpanel \((t = [150, 255])\). Three-phase loads are powered on the main panel \((t = [105, 130])\) and subpanel \((t = [255, 280])\). Indirect calibration is performed, and the results are summarized in table 3.2.

These experiments demonstrate the effectiveness of indirect calibration. The low error rate for magnitude proves the algorithm’s ability to disaggregate line currents.

Table 3.2: Calibration accuracy for delta-connected laboratory experiment.

<table>
<thead>
<tr>
<th>Meters</th>
<th>Mag. Err. [%]</th>
<th>P.F. Err. [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Known</td>
<td>Uncalibrated</td>
</tr>
<tr>
<td>Main Conv.</td>
<td></td>
<td>Sub N.C.</td>
</tr>
<tr>
<td>Main N.C.</td>
<td></td>
<td>Sub N.C.</td>
</tr>
<tr>
<td>Sub N.C.</td>
<td>Main N.C.</td>
<td></td>
</tr>
<tr>
<td>Sub N.C.</td>
<td>Sub N.C.</td>
<td></td>
</tr>
<tr>
<td>Main N.C.</td>
<td>Main N.C.</td>
<td></td>
</tr>
</tbody>
</table>
and correctly scale sensor readings. Similarly, accurate power factor verifies that
the phase rotations were correctly determined. The loads were strategically chosen
and simultaneously energized to cover most foreseeable cases. Furthermore, the
multiphase loads are ignored during calibration but considered while evaluating the
accuracy of reconstructed currents.

3.2 Shipboard Application on a Navy Vessel

As a second test environment, non-intrusive power monitors were installed aboard a
US Navy Yard Patrol (YP) vessels. This ship provides an ideal testing platform, as
generation and consumption of electricity occurs within a closed microgrid aboard
the ship. The electrical network is radially distributed around a main distribution
panel shown in Fig. 3-5. The cable entering this panel is used to deliver energy from
the generator at sea or from the utility at shore. A non-contact meter is installed
on this feeder and calibrated with the direct method to act as a source of known
$I/V$ data. The branch circuits serving various subpanels are instrumented with
uncalibrated non-contact meters.

Indirect calibration is adapted to the 208 volt delta-connected electrical system
aboard the ship, as explained in §2.6. This process was first performed for a subpanel
serving auxiliary shipboard loads. Single-phase equipment was cycled downstream
of the subpanel to produce load transients that could be measured at both the refer-
ence meter at the main panel and the uncalibrated meter at the subpanel. These test
loads included a heater and a fan that are located on different line-to-line phases.
The subpanel’s magnetic field measurements from the calibration dataset are re-
constructed into currents and processed into real and reactive components. These
results are compared against the known currents of the main panel in Fig. 3-6. After
the submeter has been indirectly calibrated, the test loads are successively energized on all phase combination again. This experiment is shown in Fig. 3-9 and validates that the calibration has succeeded. The phase isolation and scaling of transients demonstrate the correctness of the calibration matrix, while the ratio of in-phase and quadrature components verifies the phase rotations.

After calibrating the submeter, it monitored power continuously when the ship was in-port and also underway. The submetered panel serves loads that are typically used only during underway operation. An interval of the power data for both meters is shown in Fig. 3-5. The periods of high activity on the main panel correspond to daytime training exercises while the vessel is underway. These intervals match with increased power demand on the indirect-calibrated submeter panel. The total power consumption on the ship varies daily and depends on many factors, such as external

Figure 3-5: Main distribution panel aboard a YP ship (left). Power data from the indirectly-calibrated subpanel, compared to the main panel (right). Power consumption increases on both panels during underway operation.
Figure 3-6: Calibration results for the “auxiliary load” subpanel. Known currents from the main panel (left) are compared to the reconstructed currents for the subpanel (right). Calibration loads include line-to-line heaters and fans.

Figure 3-7: Test loads are energized on different phases of the indirectly-calibrated “auxiliary load” subpanel. The currents measured on the subpanel (right) all appear with the same scaling and phases rotations on the main panel (left). The B-C load on the main panel at $t = 500$ does not appear on the subpanel, since it occurs elsewhere in the electrical network.
Figure 3-8: The data from the indirectly-calibrated auxiliary subpanel is shown below the main panel power. The radar system produces pulses that are easier to detect on the subpanel. P1 and Q1 are the real and reactive power components, respectively, of the fundamental harmonic envelope.

temperature. Meanwhile, the subpanel pattern is fairly regular, corresponding to the use of key loads underway. The ship’s radar system is among these loads and is always activated while at sea. The submeter provides a definitive pulsatile indication of radar operation, as illustrated in the figure Fig. 3-8. These pulses are more difficult to identify on the main panel, which experiences transients from other loads. The radar is vital to the ship’s ability to navigate, and the submeter trace can provide important data both for confirming ship state and for diagnostic insights.

A non-contact meter is also installed on the heating ventilation and air conditioning (HVAC) subpanel. This shipboard system is automatically controlled and always running. Consequently, loads cannot be manually switched on this subpanel to produce measurable transients. High-frequency sensor readings from this uncalibrated non-contact submeter were recorded while underway, in addition to the current measurements from the main panel. The HVAC system organically produced load transients during the course of its operation. An interval of data containing line-to-line transients on two different phases was used to indirectly calibrate the submeter, resulting in the calibration shown in Fig. 3-9. To verify the correctness
of this result, the newly-determined calibration matrix and rotations were used to reconstruct the submeter currents form the sensor readings over a different time interval. This cross-validation is shown in Fig. 3-10 for more than an hour of underway operation. The load transients on the main panel and subpanel correspond in scale, phase isolation, and rotations. During both of the viewed time ranges, the HVAC system is responsible for greater than half of the power consumption of the vessel. Metering of this subsystem was previously infeasible in the shipboard environment, and indirect calibration enabled the impact of the HVAC system to be quantified for the first time.
Figure 3-9: Calibration results for the HVAC subpanel. Known currents from the main panel (left) are compared to the reconstructed currents for the subpanel (right).

Figure 3-10: HVAC subpanel data from an underway vessel. Sensor readings are retroactively reconstructed into currents (right) to cross-validate the indirect calibration of this subpanel. The currents on the main panel are shown for comparison (left).
Figure 3-11: Section of the radial power distribution network aboard the USCG Spencer. Subpanel A is located in a welding shop, and subpanel B serves receptacles throughout the ship. Yellow dots indicate the location of non-contact meters.

3.3 Shipboard Application on a Coast Guard Vessel

Non-intrusive load monitors have been deployed aboard the U.S. Coast Guard Cutter (USCGC) Spencer for diagnosing failure of critical shipboard equipment [2, 15]. This ship has a 440 volt delta-connected electrical system that is designed to sustain hundreds of amperes while underway. The main switchboard delivers power from the grid utility or onboard generators to dozens of subpanels through a radially-connected network. Indirect calibration was employed on this vessel to calibrate non-contact meters on a main power feeder and on various subpanels.

First, a non-contact meter was installed on subpanel A and directly calibrated with a line-to-line test load sequentially applied on two independent phases. Another non-contact meter was installed on the forward shore-tie cable, which delivers power to the entire vessel from the utility. The topological relationship between the uncalibrated main meter and the calibrated submeter is shown in Fig. 3-11. During this
Figure 3-12: The directly-calibrated meter on subpanel A acts as a current reference (left). The meter on the main power feed is indirectly calibrated to measure the subpanel’s load transients in addition to other shipboard loads (right).

*upstream* calibration, the test load was cycled on the subpanel to provide load transients which both meters would measure. The indirect calibration was performed on the ship’s delta-connected network using the method described in §3.2. The currents of the main power feed were reconstructed from its magnetic field measurements to verify the resulting calibration. Figure 3-12 demonstrates that the test load produced identical transients at both meters. Several critical shipboard loads were also monitored with the indirectly-calibrated main meter, as shown in Fig. 3-13.

Conventional power meters have been used to monitor various subpanels on the ship. Non-contact sensing would greatly reduce the labor and electrical hazards associated with installing submeters in this environment. However, most shipboard panels provide no means to temporarily energize a test load for direct calibration. By using the known currents at the main meter, indirect calibration can be used to calibrate a non-contact meter on any subpanel that has at least two line-to-line loads on different phases. The indirectly-calibrated main meter from the previous...
Figure 3-13: Shipboard equipment measured with the indirectly-calibrated main meter (right) and a conventional submeter (left). The main power feed exhibits fluctuations caused by other existing loads. Plots (a) and (b) show one phase of a balanced 3-phase load.
Figure 3-14: The indirectly-calibrated meter on the main feeder acts as the current reference (left). Subpanel A is indirectly calibrated to recover the transients of the test load (right).

The experiment was used to recalibrate submeter A using the same meters and test load as before. This downstream calibration is symmetrical to the previous experiment, and the resulting currents are expected to reproduce the test load’s transients on the subpanel, as shown in Fig. 3-14. Since the result of the upstream calibration is used as reference data for the downstream calibration, the amount of error is amplified at each of these interdependent experiments and results in slight cross-talk on phase B. Additionally, accurately measuring the current transients at the main meter is difficult due to other active loads in the system producing fluctuations over steady-state intervals.

A similar set of experiments were conducted using subpanel B, which branches from the main switchboard in Fig. 3-11. A non-contact meter was installed on this panel and indirectly calibrated against the main meter calibrated in a previous experiment. The downstream calibration shown in Fig. 3-15 was able to accurately reconstruct the subpanel current transient. These experiments jointly demonstrate...
Figure 3-15: The indirectly-calibrated meter on the shore-tie acts as the current reference (left). Subpanel B is indirectly calibrated (right).

Figure 3-16: The meter on subpanel B was directly calibrated to provide reference currents (left). The main main was indirectly calibrated to yield the reconstructed currents (right).
that an existing submeter can be used to indirectly calibrate a new subpanel by using a common upstream point in the system.

3.4 Conclusion

Mission-critical systems in industrial, commercial, and military environments may operate with little or no opportunity for scheduled downtime. Power system diagnostics that could be used for fault detection, diagnosis, and prognosis may be unavailable, therefore, unless originally installed with these systems. When retrofit or temporary power monitoring is desirable for energy scorekeeping, activity tracking, or diagnostic monitoring, non-contact power meters can acquire power information without the need to break a wire or make ohmic contact with a conductor.

This paper has introduced an indirect calibration algorithm for use with non-contact power monitoring sensors. This algorithm makes it possible to install and calibrate non-contact power meters even when the electrical loads cannot be disturbed, altered, or added to for inclusion of a calibration load. Indirect calibration permits power data from any other convenient meter at a site to provide a calibration reference for a new meter. This new meter can be installed upstream or downstream on a radial power distribution network for a “wide angle” or “zoomed in” view of a collection of loads.
Modern ships rely on a range of automated systems to operate critical machinery, which can fail suddenly due to a lack of prognostic indicators. A non-intrusive load monitor (NILM) can provide indicators for a collection of equipment by monitoring their electrical loads from a single point in the shipboard power system. Preventative maintenance is often time-sensitive, and equipment may fail before NILM data can be post-processed.

The NILM Dashboard is a machine intelligence and graphical platform that provides information about equipment in real time to non-technical crew members. The operation of individual loads is disaggregated using signal processing and presented as time-based load activity and statistical indicators. A graphical user interface provides analysis tools for detecting fault conditions, determining operating state, and energy scorekeeping. The software allows multiple NILM devices to be networked together to provide information about loads residing on different electrical branches at the same time. The NILM Dashboard is demonstrated on the power system data from two USCG Cutters.
4.1 Introduction

Crews sizes aboard commercial and military ships have consistently decreased over the past several decades. This shift corresponds to an increased use of automated systems to perform critical shipboard functions. As a result, the compact crew is responsible for monitoring and maintaining a substantial set of machinery \[14\].

Present-day monitoring systems employ expansive sensor networks that are costly to install and experience sensor failures as regularly as the equipment they are intended to protect \[27\]. Instead of measuring mechanical indicators directly, the electric power system can be used to monitor mission-critical electromechanical equipment for diagnostic and prognostic indicators of system health \[2, 15\].

The flow of energy and information between producer and consumer is also crucial to the emergence of the Smart Grid \[28\]. Consistent feedback on consumption allows power generation to be adjusted for rapid changes in loading. Increased knowledge about energy usage can reduce individual demand and identify malfunctioning appliances. The pipeline for this information flow requires an infrastructure of advanced metering, monitoring and data management \[29\]. This infrastructure must be created without large scale grid disruptions and the information gathered must be readily available and actionable. A Non-Intrusive Load Monitor (NILM) is a rugged, low-cost sensing device that can fill this need.

A NILM may be installed at the main power entry or at a subpanel to measure the operating schedules and diagnostic conditions of a collection of electrical loads \[1\]. The device can measure currents and voltages with a conventional power meter or with non-contact sensors that do not require ohmic contact \[12\]. Electrical data is stored in a high-speed time-series database \[8\] and converted into harmonic power envelopes to facilitate analysis \[9\]. A NILM uses signal processing to detect
the operation of individual loads from the aggregate power data. Software tools can be used for algorithmically post-processing the electrical measurements for disaggregation, energy score-keeping, and diagnostics [10]. This analysis software is richly featured and can output results immediately; however, these tools are best suited to technical users and not specialized for a shipboard environment.

In practice, a NILM may capture electrical data as a “black box” until retrieved for analysis. In commercial, industrial, and naval environments, mission-critical equipment can abruptly malfunction. Symptoms are often visible in the electrical system weeks before a failure occurs [3].

The NILM Dashboard can track the operation of critical equipment and predict faults before they become catastrophic. Prognostic indicators are communicated to operators in a matter of hours or days, so that preventative maintenance can be performed early. The system provides a user interface for operators to analyze condition and use patterns in real time. This software functionality can be added to an existing NILM.

The NILM Dashboard adds advanced diagnostic and operational tracking capabilities to a NILM system, demonstrated here on two 270-ft US Coast Guard Cutters (Fig. 4-1). The system retains favorable aspects of NILM – low sensor count, easy installation, high reliability – but expands NILM capability to provide support for condition-based maintenance, fault detection, and diagnostics. On each ship, the system monitors two electrical subpanels in the main engine room, as well as the overall power consumption of the vessel. The subpanels power equipment crucial to the proper operation of the engineering plant. The NILM Dashboard is a shipboard automated watchstander (SAW), delivering maintenance and fault data to Coast Guard personnel about shipboard loads in real-time, ultimately reducing equipment failures and crew fatigue [17][18]. This paper presents the NILM Dashboard and its
4.2 System Architecture

The NILM Dashboard software stack runs on a typical Linux-based computer. The 4 layers shown in Fig. 4-2 are responsible for obtaining measurements, identifying loads, analyzing behaviors, and communicating results. The Dashboard processes information about load events, which occur when equipment transitions between ON and OFF states. The event data is classified as a specific load and then mapped to an operating schedule of its activity. This time-based information enables detailed comparison of loads and provides a record of operations. To highlight trends over time, the platform also maintains load metrics, which are statistical conclusions that expose anomalies and patterns.
4.2.1 Data Capture & Pre-Processing

The NILM captures currents and voltages from metering hardware. These readings are pre-processed into harmonic envelopes using the sinefit algorithm, which computes real and reactive power components at the line frequency and at higher-order harmonics [9]. Sinefit outputs a single data point per line cycle, promoting space-efficiency while maintaining the richness of the original signals. NilmDB [8] is specially suited for storing this time-series data, making it available for high-speed and low bandwidth access throughout the dashboard platform.
4.2.2 Load Identification

The load identification block disaggregates the operation of individual loads from the power data, as explained in §4.4. The resulting load events encode the type of state transition, as well as the change in real and reactive power levels. NilmDB [8] records these sparse events at the times they occur. The Joule data processing framework [31] is used to streamline the capturing, pre-processing, and identification stages in Fig. 4-2. This robust tool models the data pipeline as a series of processing “modules”, with formally-defined “streams” of information passing between them. Data flows between modules through efficient memory pipes, without needing to access the database as an intermediary. However, the pre-processing and identification stages each copy their outputs to NilmDB for later use.

4.2.3 Metrics Generation

The “Metric Generator” is responsible for reading event streams from NilmDB and calculating metrics, which are operational statistics over a given time period. Relevant metrics for a load may include the average power consumption or the number of operations per day. The Dashboard must be able to rapidly query metrics using any of these parameters. Each metric has metadata about its type, the load, and the encompassed time range. A NoSQL database structure is employed for the “Metrics DB” [32, 33] due to its easy expandability compared to traditional SQL alternatives. The Dashboard tracks an initial set of parameters, but these are expected to expand with increased knowledge of the system and loads. Whereas the Metrics DB can store any number of metrics for a given load, a table-based structure would not easily adapt beyond the initial set of metrics. The Metrics DB also has the ability to save large data segments at full resolution, allowing for an anomalous load transient.
to be stored alongside its respective metrics. An SQL database enforces stricter size boundaries and may need to preallocate a fixed amount of space.

The Metric Generator is implemented as a Python-based script that runs automatically every ten minutes. Every time the generator runs, it calculates rolling metrics over the past 24-hour window. These values are used to update the gauges in the metrics interface (§4.3.2) to reflect current conditions. The Metric Generator also calculates daily load metrics if event data is available in NilmDB for the previous day, which ends at midnight. This information is used to populate the historic view (§4.3.3) of the user interface, which shows the daily progression of a chosen metric for a given load.

4.2.4 Graphical Interface

The user interacts with a web-based application [34, 35] that is served from the NILM. This can be accessed on the Dashboard computer or from any computer connected on the same network. The user interface operates exclusively in the client’s browser, conserving processing power at the NILM and network bandwidth. The data visualizations are implemented with a data-driven graphical framework [36].

The browser must retrieve event and metric data to populate the interfaces. NilmDB has external web endpoints, so the website can directly communicate with this database to download event data for a particular load. The metrics database, however, cannot be accessed outside of the NILM. The web server proxies requests to the metrics database by exposing its own web endpoints for the client to reach.
4.2.5 Networked Data Expansion

In many environments, a single NILM is inadequate for monitoring the entire electrical system. Figure 4-3 shows NILM monitors at various points aboard a USCG Cutter. Two conventional “Contact” meters are installed on the Port and Starboard subpanels to provide high resolution measurements of mission-critical equipment. The non-contact meters installed at the main power feeds measure total vessel power consumption but with lower resolution of individual loads. Each NILM provides unique diagnostic insights individually and can be used to make conclusions jointly. Both NILM monitors are necessary for complete system analysis, but are over a hundred feet apart within the vessel. The Dashboard software provides database access on a local network for the NilmDB and Metrics DB. Therefore, a user can access all
the other NILM nodes on the local network from one place.

4.3 User Interface

The volume of NILM research does not address the challenge of making NILM data actionable and available for operators. Providing the information generated by NILM quickly and clearly is paramount to creating a diagnostic tool capable of preventing failures to mission critical equipment. The NILM Dashboard addresses this problem with an on-site interface that provides real-time and historic equipment information. To further aid operators in tactical decision making, the Dashboard generates useful metrics on system health. This information is made available to operators through an interface containing four interactive tools: Timeline, Metric View, Historic View, and Loading View.

4.3.1 Timeline

The Timeline, as shown in Fig. 4-4 provides a live view of the equipment status, allowing the user to see loads activated and secured in real time. The user can monitor the entire plant or hide certain equipment from the view, allowing for increased attention on select loads. The time window can be adjusted to the user’s choosing, either through zoom/pan functions or by selecting one of four pre-set time periods. The Timeline tool provides the user with a compact picture of plant operations and the ability to easily investigate any apparent anomalies.
4.3.2 Metric View

The Metric View shown in Fig. 4-5 is a user’s first stop when a fault is suspected. It provides the user with a set of diagnostic indicators for a selected piece of equipment. The metrics available are real power, apparent power, power factor, average run duration, total daily run time and daily number of actuations. Each metric is displayed as a gauge with green, yellow and red sections. The colored sections are derived from equipment nameplate data, known usage patterns and statistics from previous normal operation. Green indicates normal operations, while yellow and red indicate increasing likelihood of a fault or casualty. The gauge needle position is the average metric value for the last 24 hours and is refreshed every ten minutes. The Metric View provides an analysis of individual equipment health and helps direct initial troubleshooting efforts.

Figure 4-4: The timeline interface displays when each load was energized.
### 4.3.3 Historic

The Historic view shown in Fig. 4-6 provides short and long-term trend data to supplement the analysis from the Metric View. This tool allows users to select a single load and any one of the six metrics listed above. The Historic View is presented as a bar graph that gives the user the ability to track equipment behavior over a period of up to 6 months. While the metric view is intended for a watchstander to quickly detect a possible casualty, the historic view is designed for the plant manager to assess trend data, track behavior and make decisions on condition-based maintenance.

### 4.3.4 Loading View

The Loading View shown in Fig. 4-7 allows users to detect phase imbalances and loading discrepancies within the electrical system. The user can select a monitoring point and view the per phase power and total electrical load. For instance, in this case study the Loading View allows the user to compare the total loading of the two generators or the two monitored subpanels. This information can be used for
energy-scorekeeping and to optimize power generation.

4.4 Load Identification

Load identification is a key step in the four-stage pipeline. Load identification can be accomplished by many different algorithms, such as such as artificial neural networks (NN), k-nearest neighbors (k-NN), and correlation-based algorithms $[8,37]$. This is where the core of NILM research is focused today. For this application, a neural network approach is taken. Since the two monitored subpanels have a fixed number of loads, a supervised learning approach is used in which data is hand-labeled in order to perform training. As described by Hart $[1]$, there are three main categories of appliances that may be monitored by a NILM system: ON/OFF, Finite State Machine (FSM), and continuously variable. On the studied subpanels there are ON/OFF loads and one FSM load. An ON/OFF load has only two states, ON or
Figure 4-7: The loading view displays the currents per-phase levels for a source of power data. With the shown configuration, the port and starboard subpanel are being compared.

OFF, while a FSM load has several states due to its complex operation. Table 4.1 lists all the monitored loads which are to be displayed on the Dashboard. The following sections describe how the NILM Dashboard system disaggregates a continuous power stream into a list of discrete, classified events.

Load identification occurs in the following three stages:

**Event Detection** Determines when transients occur.

**Event Classification** Matches identity of transient to a load.

**Load Confirmation** Checks constraints between load events.

### 4.4.1 Event Detection

An edge detector locates load turn-on and turn-off times by identifying when a the apparent power ($S$) stream has abruptly changed in value. Apparent power ($S$) is
<table>
<thead>
<tr>
<th>Load</th>
<th>Power Rating</th>
<th>Delta Phases</th>
<th>Power Factor</th>
<th>Port Panel</th>
<th>Stbd Panel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main diesel engine (MDE) keep-warm system</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lube oil (LO) heater</td>
<td>12 kW</td>
<td>3ϕ</td>
<td>1.0</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Jacket water (JW) heater</td>
<td>9.0 kW</td>
<td>3ϕ</td>
<td>1.0</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Prelube pump</td>
<td>2.2 kW</td>
<td>3ϕ</td>
<td>0.82</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td><strong>Ship service diesel generator (SSDG) keep-warm system</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jacket water (JW) heater</td>
<td>7.5 kW</td>
<td>3ϕ</td>
<td>1.0</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Lube oil (LO) heater</td>
<td>1.3 kW</td>
<td>1ϕ</td>
<td>1.0</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td><strong>Additional engine room loads</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Controllable pitch propeller (CPP) hydraulic pump</td>
<td>7.5 kW</td>
<td>3ϕ</td>
<td>0.82</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Graywater pumps</td>
<td>3.7 kW</td>
<td>3ϕ</td>
<td>0.85</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Diesel oil (DO) purifier</td>
<td>5.6 kW</td>
<td>3ϕ</td>
<td>0.85</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Oily water separator</td>
<td>6.7 kW</td>
<td>3ϕ</td>
<td>0.90</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
given by,

\[ S = \sqrt{P^2 + Q^2} \]  

(4.1)

where P is real power and Q is reactive power, which are produced by the sinefit algorithm \([9]\). Converting to apparent power simplifies load detection to a single data stream. Apparent power is a reliable source for detecting energy changes in a controlled environment with a known collection of loads. This stream is then smoothed using a 101-point median filter, which eliminates small fluctuations while preserving edges. The filtered data is convolved against the Laplacian of a Gaussian \([38]\) kernel to compute the smoothed second-derivative of this stream. An empirically-determined threshold is set to remove zero-crossings that are due to small fluctuations of the resulting convolution. A zero-crossing detector is then used to find the location of the steps. Fig. 4-8 shows several ON and OFF events as detected by the edge detector.
After an ON or OFF event is detected, a set of features is calculated for each phase to be used as an input vector to the NN. An ON event produces four features for a single phase, a transient-peak ($P_{on,x,peak}$, $Q_{on,x,peak}$) and a steady-state level change ($P_{on,x,peak}$, $Q_{on,x,peak}$). Because there is no transient peak when a load is secured, an OFF event has only two features per phase ($P_{off,x,ss}$, $Q_{off,x,ss}$). The steady state level change is calculated by taking the difference in the median power level for a defined window ($\Delta t_{window}$) before and after an ON/OFF event. The $\Delta t_{window}$ is chosen to be 0.5 sec, which is 30 samples given a line frequency of 60 Hz. Fig.4-9 illustrates these features for a single phase. Equations (4.2)-(4.6) generate the features and the input vectors.

ON event steady state (ss) change in power:

$$P_{on,x,ss} = median[P_x(t_{end}), ..., P_x(t_{end} + \Delta t_{window})] - median[P_x(t_{on} - \Delta t_{window}), ..., P_x(t_{on})]$$

(4.2)

ON event transient peak:

$$P_{on,x,peak} = max[P_x(t_{on}), ..., P_x(t_{end})] - median[P_x(t_{end}), ..., P_x(t_{end} + \Delta t_{window})]$$

(4.3)

OFF event steady state (ss) change in power:

$$P_{off,x,ss} = median[P_x(t_{off}), ..., P_x(t_{off} + \Delta t_{window})] - median[P_x(t_{off} - \Delta t_{window}), ..., P_x(t_{off})]$$

(4.4)

where $x$ is the phase (A,B,C), $t_{on}$ is the time the load turns on, $t_{end}$ is the end of the
start-up transient, \( t_{\text{off}} \) is the time the load turns off, and \( t_{\text{window}} \) is the length of the window for taking steady state calculations.

The feature input vector for ON events is:

\[
\begin{align*}
(P_{on,A,ss} \ Q_{on,A,ss} \ P_{on,A,peak} \ Q_{on,A,peak} \\
P_{on,B,ss} \ Q_{on,B,ss} \ P_{on,B,peak} \ Q_{on,B,peak} \\
P_{on,C,ss} \ Q_{on,C,ss} \ Q_{on,C,peak} \ Q_{on,C,peak})
\end{align*}
\] (4.5)

The feature input vector for OFF events is:

\[
\begin{align*}
(P_{off,A,ss} \ Q_{off,A,ss} \\
P_{off,B,ss} \ Q_{off,B,ss} \\
P_{off,C,ss} \ Q_{off,C,ss})
\end{align*}
\] (4.6)

There are a fixed number of loads on the panels and the normal real and reactive power draw of each load is known. Therefore the total change in steady state should not be less than the smallest load on the panels. False detections are reduced by comparing the magnitude of the calculated change in steady state values to a threshold. Changes beneath this threshold are discarded. Using one month of data from September 1, 2017 to September 30, 2017 on the USCGC Spencer port panel, there were a total of 2946 events. The described edge detector correctly found 2936 of those events. The detector missed 10 events and incorrectly detected 32 non-events.

### 4.4.2 Event Classification

To classify each event as an individual load, a fully-connected neural network (NN) using stochastic gradient descent (SGD) is applied \[39\]. Separate NNs are utilized
for ON events and OFF events for each panel. The input layer to the NNs are the features for each load as previously described in (4.5) for ON events and (4.6) for OFF events. Two hidden layers of 10 nodes each are used, for which a weighted sum of the inputs from the previous layer to each node is passed through a ReLu activation function, $f(z) = \max(0,z)$, where $z$ is the input to a node. Back-propagation is used to find the set of weights and biases to minimize loss. The output layer is a softmax layer, defined as,
\begin{equation}
    f(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \text{ for } i = 1...k
\end{equation}

where $z$ is the input to each node, and $k$ is the number of classes. Each ON/OFF load is a unique class. Additionally, some loads may actuate together during normal operations, creating a new class representing multiple loads. Finally, a FSM load such as the Diesel Oil Purifier will have a class for each state of operation. Table 4.2 displays the classes used on the NN. A softmax output layer is used because it enables multi-class classification. The output is a vector of probabilities that sums to 1 and the classification is made by selection of the class that has the highest probability.

Data from one month (September 1, 2017 to September 30, 2017) on the USCGC Spencer port panel is split into three data sets: training, validation, and testing. The prediction error of the validation set is used as a stopping criterion during training [40]. To prevent over-fitting, a third separate data set, the testing set is used. Repeated random sub-sampling is performed on the data set for 10 iterations, so that for each iteration the data is randomly split into training, validation, and testing data and the performance is evaluated on the testing data.

Some loads have short runs, in the range of minutes, and thus have many events. Other loads have run durations in the range of hours or even days, and thus will have far fewer events. To prevent the NN from simply predicting the most common class, the training data needs to be better balanced. This is accomplished by random under-sampling of the majority classes on the training and validation data. It is not sufficient to focus only on the total percentage of correctly classified loads when verifying the accuracy of the identifier. If a load only turns on and off a few times over a month, incorrectly identifying it will not have much effect on the total classification accuracy, but it is still vital that the load be correctly identified. Thus, the accuracy
of the model for each class is verified by considering the following parameters [41]:

- True Positive (TP): load event occurs and is correctly identified
- False Positive (FP): no event occurs but an event is identified
- False Negative (FN): load event occurs but is incorrectly identified

These parameters are used to determine the model’s recall and precision, which answer two fundamental questions about the model:

1. Do load events get reported? (recall)
2. Are load events reported correct? (precision)

\[
\text{recall} = \frac{TP}{TP + FN} \quad (4.8)
\]
\[
\text{precision} = \frac{TP}{TP + FP} \quad (4.9)
\]

Table 4.2 shows the results of the average of 10 iterations of the model processing a dataset from the USCGC SPENCER. The results presented are the number of true positives, precision, and recall for each class, as well as the standard deviations of each (\(\sigma_{TP}, \sigma_{P}, \text{ and } \sigma_{R}\) respectively). Precision and recall values of "1" with a standard deviation of zero indicate perfect performance in identifying a specific class. For true positives, a small standard deviation shows that performance between iterations is consistent and that the model is not overfitting.

### 4.4.3 Confirmation and Implementation

Load identification is implemented into the NILM Dashboard architecture as shown in Fig. 4-2 through two modules, classification and confirmation. The classification
<table>
<thead>
<tr>
<th>Class</th>
<th>TP ± σ&lt;sub&gt;TP&lt;/sub&gt;</th>
<th>precision ± σ&lt;sub&gt;P&lt;/sub&gt;</th>
<th>recall ± σ&lt;sub&gt;R&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main diesel engine (MDE) keep-warm system</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LO Heater</td>
<td>3 ± 0</td>
<td>1 ± 0</td>
<td>1 ± 0</td>
</tr>
<tr>
<td>Prelube Pump</td>
<td>2.1 ± 1.20</td>
<td>0.84 ± 0.35</td>
<td>.52 ± .30</td>
</tr>
<tr>
<td>LO Heater and JW Heater</td>
<td>0.9 ± .32</td>
<td>1 ± 0</td>
<td>.9 ± .32</td>
</tr>
<tr>
<td>LO Heater and Prelube Pump</td>
<td>4 ± 0</td>
<td>.96 ± .08</td>
<td>1 ± 0</td>
</tr>
<tr>
<td><strong>Ship service diesel generator (SSDG) keep-warm system</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JW Heater</td>
<td>65 ± 0</td>
<td>1 ± 0</td>
<td>1 ± 0</td>
</tr>
<tr>
<td>LO Heater</td>
<td>95 ± 0</td>
<td>1 ± 0</td>
<td>1 ± 0</td>
</tr>
<tr>
<td><strong>Additional engine room loads</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPP Pump</td>
<td>4.5 ± 1.1</td>
<td>.90 ± .11</td>
<td>.90 ± .22</td>
</tr>
<tr>
<td>Graywater Pump</td>
<td>267.9 ± .31</td>
<td>1 ± 0</td>
<td>1 ± 0</td>
</tr>
<tr>
<td>DO Purifier (Centrifugal)</td>
<td>26.3 ± .67</td>
<td>0.98 ± .04</td>
<td>0.97 ± .03</td>
</tr>
<tr>
<td>DO Purifier (Feed Pump)</td>
<td>6.4 ± 1.90</td>
<td>0.90 ± .14</td>
<td>0.91 ± .27</td>
</tr>
<tr>
<td>DO Purifier (Feed Pump)</td>
<td>14 ± 0</td>
<td>1 ± 0</td>
<td>1 ± 0</td>
</tr>
</tbody>
</table>
module runs the previously described detection and classification algorithm on 10 second windowed power stream data. Since there may be events that occur at the edges of an interval, the intervals are overlapped to ensure that no events are lost. For each iteration, the classification module output is piped to a confirmation module. For each load, the confirmation module stores the most recent state (ON or OFF) outputted to NilmDB, to ensure two ON events or two OFF events are not outputted consecutively. If two consecutive ON’s are detected, the confirmation module removes the first occurrence from the NilmDB. If two consecutive OFF’s are detected, the second occurrence is not outputted to the NilmDB.

For greater classification accuracy, there is further research being done on integrating this NN method with other classification methods such as an exemplar shape-matching algorithm [8] and multi-scale median filtering. This would all be done within the classification module, allowing the classification algorithm to be updated without affecting the operation of the user interface.

4.5 Experimental Results

The NILM Dashboard is designed to consume electrical readings from a hardware meter and process these signals into event streams. Manually cycling equipment on an in-port ship provides load transients that can be used to validate the load identification process. However, this controlled experiment will not authentically simulate the scale and entropy of an underway dataset. Since a pair of NILM devices recorded the port and starboard subpanels, this data could be retroactively injected into the load detector as if it were an active meter.

To validate each layer of the Dashboard architecture, the system was bench-tested on numerous busy intervals of the underway data. The Load Identification
yielded a high rate of accuracy. To assess the system end-to-end, the identification results were viewed via the timeline interface. The generated load events are shown for a short, quiet period of data in Fig. 4-10. The load transients appearing in the original data stream have been time-aligned with the events generated. The identity of the loads can be visually discerned in this simple example. Figure 4-11 shows an 11-hr period of data which contains the interval from Fig. 4-10. The dataset contains two high-energy loads that experience fluctuations. In this end-to-end test, the load identification algorithm successfully detected the ON and OFF changes.

A yet more complex situation is present in Fig. 4-12, which shows longer dataset that encompasses a boarder variety of loads.

4.6 Conclusion & Future Work

The NILM Dashboard software can be installed onto a new or existing NILM to provide prognostic insights about shipboard equipment, without the need for an extensive sensor network. The platform has been shown to successfully detect and display the operation of a variety of loads from underway electrical data. The system is poised to be installed aboard a Coast Guard vessel to monitor load activity in real time.

Once the user interface is made available to the crew, the Dashboard must earn their trust by reporting load events that can be visually confirmed. In time, the tool can be relied upon to improve the crew’s awareness of faults in shipboard systems. Future software development efforts are planned to include automated analysis of event patterns and statistics. By supplementing human intuition with algorithmic logic, the Dashboard will be able to alert crew members of potential issues before they are noticed.
Figure 4-10: The timeline (below) displays the disaggregated events from the gray-water pump (GW) and a lube oil (LO) heater over a 4-hr period. The real (P) and reactive (Q) power components for each phase are shown for comparison (above).

Figure 4-11: The dataset (above) includes two high-energy loads – DOP and SSDG JW – that correspond to two uninterrupted activations on the timeline.
Figure 4-12: Load events are shown over a 2-day interval. All the transients present in the origin signals (above) have been detected and are shown in the timeline (below).
Chapter 5

Bibliography


95


Appendix A

Instructional Guides

A.1 Performing Indirect Calibration

A non-contact meter can be calibrated against an existing meter in the electrical network, as explained in chapter 2. The following instructions serve as a practical guide to performing indirect calibration in the field.

A.1.1 Prerequisites

An uncalibrated meter is installed at a new point within the electrical network. Suitable test loads must be located downstream of this meter which energize independent phases. In a wye-connected system, there must be three loads – on the A, B, and C phases; in a delta-connected system, there are only two loads required – on any two A-B, B-C, or A-C phases. A calibrated meter must exist within the electrical network that measures the currents passing through the uncalibrated meter. The calibrated meter may be a conventional contact-based meter or a non-contact meter that has already been calibrated.
A downstream calibration occurs when the uncalibrated meter measures a subset of the current measured by the calibrated meter, such as when installing a submeter. Transposing the position of these meters would constitute an upstream calibration. The calibration process is identical for either of these situations.

If the meters are co-located, they may be connected to the same computer. Otherwise, the measurements must be acquired using two separate computers. The following requirements must be met on each computer:

- All other capturing utilities must be stopped. This includes nilm-capture, joule, and nilm-scope.
- NilmDB must be running.
- The meter details information must be in the configuration file located at /opt/configs/meters.yml.
- The system clock must be synchronized within ±60 seconds between machines.

The nilm-submeter command must be linked to the main submeter.py file of the source code in appendix C.1.

### A.1.2 Capturing Signals

**Capture Currents from Calibrated Meter**

Capture the currents at the calibrated meter with the following command. The text meterX should be substituted with the meter name. The output file will be located in the current directory and named curr.dat in this example.

```
nilm-submeter capture -m meterX -c curr.dat
```
Capture Sensor Readings from Uncalibrated Meter

Simultaneously, capture the sensor readings at the uncalibrated meter with the following command. Note that \texttt{meterY} is a different meter than before. The output file will be located in the current directory and named \texttt{sens.dat} in this example.

\begin{verbatim}
  nilm-submeter capture -m meterY -s sens.dat
\end{verbatim}

Cycle Loads and Finish Capturing

Cycle the chosen loads on each phase several times. To produce detectable load transients, wait a few seconds between switching loads on and off. To finish capturing, type \texttt{STOP} as instructed on-screen on each computer. The measured signals will be processed and written to their respective output files.

View Captured Signals

The captured currents have already been processed into PREP streams. View the captured currents at the calibrated meter with the following command, which will display a plot like the one shown in Fig. A-1.

\begin{verbatim}
  nilm-submeter show -c curr.dat
\end{verbatim}

The captured sensor readings have not been processed and are saved as high-frequency signals. Zooming in will reveal the sinusoidal waveforms of the magnetic and electric fields. View the captured sensor readings at the uncalibrated meter with the following command, which will display a plot like the one shown in Fig. A-2.

\begin{verbatim}
  nilm-submeter show -s sens.dat
\end{verbatim}
Figure A-1: Calibration utility displaying captured currents from the calibrated meter.

Figure A-2: Calibration utility displaying captured sensor readings from the uncalibrated meter.
Figure A-3: Current plot produced upon completed calibration process. Known currents for each phase of the previously-calibrated meter are shown in the top three rows. Currents reconstructed from the newly-calibrated meter are shown in the bottom three rows.

Figure A-4: Sensor scales visualization produced upon completed calibration process. Clusters of blue dots represent load transients. The best-fit line (red) is fit along the transient points.
A.1.3 Calibrating

The calibration process occurs on the computer connected to the uncalibrated meter. Copy the current file curr.dat onto this computer, which should already have the sensor measurements file sens.dat. The calibration command below should be invoked with these files and the uncalibrated meter name in place of meterY.

```
nilm-submeter calibrate -m meterY -c curr.dat -s sens.dat
```

In most cases, the calibration process will complete successfully and without intervention. The program will produce various plots of the resulting calibration. The currents from the reference meter are plotted against those reconstructed from the magnetic fields on the newly calibrated meter, as shown in Fig. A-3. The sensor scales relating each magnetic field sensor to each current phase are also plotted, as shown in Fig. A-4.

A.1.4 Debugging and Adjusting

In rare cases, the calibration process may return a textual error or produce incoherent results. The program is designed to print an error if there are not enough load transients detected on each independent phase to complete the calibration. This may indicate a physical issue, in that an insufficient number or type (ex. 3-phase) of loads transients were captured. However, it may also indicate that the transient detector was unable to locate these changes.

Adjusting the Transient Detector

The submeter calibrate command can be executed with a --plot-trans flag to expose the intermediate signals within the transient detector. As each transient stream is generated – for each current and sensor stream – a plot will appear as
Figure A-5: Intermediate streams used to adjust transient detector. The shown “Transient Stream” results from using the detector with the displayed thresholds.

shown Fig. A-5. The “Transient Stream” results from applying the detector with the displayed thresholds. These thresholds should be adjusted for the algorithm to detect physically relevant transients. The value for $corr\_thresh$ should be selected so that all transients produce a value $greater$ than this threshold on the “Volatility Correlation” stream. The value for $balance\_thresh$ should selected so that all transients produce a value $less$ than this threshold on the “Balance” stream. The adjustment arguments are listed below. For instance, \texttt{--param-tcc 50.0} can be added to the calibration command to set $\tau_{corr} = 50$.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Parameter</th>
<th>Streams Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{--param-tcc}</td>
<td>$corr_thresh (\tau_{corr})$</td>
<td>Currents</td>
</tr>
<tr>
<td>\texttt{--param-tbc}</td>
<td>$balance_thresh (\tau_{bal})$</td>
<td>Currents</td>
</tr>
<tr>
<td>\texttt{--param-tcs}</td>
<td>$corr_thresh (\tau_{corr})$</td>
<td>Sensors</td>
</tr>
<tr>
<td>\texttt{--param-tbs}</td>
<td>$balance_thresh (\tau_{bal})$</td>
<td>Sensors</td>
</tr>
</tbody>
</table>
Verifying Transients and Alignment

Once the transient detector has been tuned, the calibration process can be re-attempted with these adjustments. To view the transient streams for all currents and sensors at once, the --plot-diffs flag should be added to the calibration command. An example of this plot is shown in Fig. A-6. The streams have already been aligned, so transients pertaining to the same physical event should appear at the same instant across all streams. Note that the calibration algorithm is tolerant to transient misalignment by ±5 samples between streams.

If the transients do not appear to match up, then it may be necessary to ensure that the streams have been aligned correctly. The intermediate streams of the alignment process can be plotting using the --plot-align flag. Figure A-7 shows the magnitudes of the current and sensor streams before they have been aligned, which can be visually inspected to determine the correct time offset. The “Offset Correlation” plot shows the amount of correlation between the currents and the sensors at each time offset. Its peak corresponds to the desired time offset, when the streams are most similar.
Figure A-6: Aligned transients on current and sensor streams.

Figure A-7: Intermediate streams for the alignment process. The current and sensor magnitudes are shown before alignment has occurred. The “Offset Correlation” peaks at the time offset when the streams are most similar.
A.2 Creating a Nilmbuntu Installer from Source

The NILM Suite is a collection of software tools for acquiring, managing, and processing power data. At the time of writing, it includes the following programs:

- ethstream
- flot
- ljstream
- nilm-calibrate
- nilm-consume
- nilm-manager
- nilmcapture
- nilmdb
- nilmplug
- nilmrun
- nilmtools
- noncontact_firmware
- t7-capture
- submeter

The NILM Suite can be installed onto an existing Linux machine, such as Ubuntu version 16.04, using the “standalone installer”. It can also be installed automatically with the “Nilmbuntu” operating system [42]. Nilmbuntu is a Xubuntu-based operating system with all NILM software pre-installed. Nilmbuntu can be installed graphically from a bootable flash drive by a non-technical individual, just like most modern Linux installations. This guide explains how to create the Nilmbuntu installation media from its software source.

A.2.1 Machine Prerequisites

This section only needs to be completed ONCE on the development machine.

The “Stick Maker” machine provisioned for use with this guide already has the required software and repository access permissions.

1. Install a clean version of Ubuntu 17.04 Desktop Edition or later. This process requires a newer version of systemd, which is unavailable in older Linux in-
stallations. The alternative is to patch the system kernel, which is tedious and can result in an unstable environment.

2. Install system dependencies by running the following commands:

```
sudo apt install git
sudo apt install squashfs-tools xorriso isolinux iptables
sudo apt install systemd-container qemu
sudo systemctl start systemd-networkd
```

3. Create an account on the Wattsworth GitLab Site\[1\] Obtain permission from John Donnal (donnal@usna.edu) for the nilm and wattsworth groups. This will allow access to the necessary NILM Suite software repositories. Link the machine to the GitLab account using SSH keys.

### A.2.2 Preparing the NILM Suite

1. Navigate to the home directory:

   ```
   cd ~
   ```

2. Obtain the “Standalone Installer” repository. This utility installs all the NILM packages onto an existing Linux OS.

   ```
   git clone git@git.wattsworth.net:nilm/standalone_install.git
   ```

3. Retrieve the source code for all the individual NILM software repositories.

   ```
   cd standalone_install
   git submodule update --init --recursive
   ```

---

\[1\]https://git.wattsworth.net
4. Retrieve all external dependencies, which do not qualify as source code.

   ./get_dependencies.sh

### A.2.3 Building the Installation

1. Navigate back to the home directory:

   ```
   cd ~
   ```

2. Obtain the “Nilmbuntu” repository. This utility allows the developer to manipulate the Nilmbuntu OS before it is packaged into a bootable installer.

   ```
   git clone git@git.wattsworth.net:nilm/nilmbuntu.git
   ```

3. Download and extract the original Xubuntu OS. *(Check Nilmbuntu README for current version.)*

   ```
   cd nilmbuntu
   export BUILD_CONFIG=16.04.2
   ./extractiso.sh
   ```

4. Apply customizations to the Nilmbuntu system, including installing dependencies and changing configurations.

   ```
   ./customize.sh
   ```

5. Copy the NILM Suite Installation into the Nilmbuntu system.

   ```
   sudo cp -r ~/standalone_install tmp-16.04.2/fs/root/
   ```

At this point, there is an “outer” and “inner” system. The “outer” machine is the physical development machine, which has executed all the commands thus far. The
“inner” machine is the Nilmbuntu system that is being modified. The next few steps enter into the inner scope, make changes, and then return to the outer scope.

It is important to be aware of the scope transition, so that the modification are not applied to the development machine itself.

6. Enter into the inner Nilmbuntu system. The command prompt changes to

```
root@nilmbuntu:~#
```

```
./enter.sh
```

7. Install the NILM Suit into the inner system. The standalone installer report

```
Install Complete
```

upon successful completion.

```
cd standalone_install
```

```
./install-iso.sh
```

8. Exit from the inner system. The prompt changes back to [user]@[machine]:~/nilmbuntu

```
exit
```

9. Build the modified system into an installation image. The image file will be named nilmbuntu-16.04.2.iso.

```
./buildiso.sh
```

**A.2.4 Testing and Deploying**

**Bootable Disk**

The image can be written to a flash drive and used as a bootable installer.

1. Insert the flash drive and run `lsblk` to list all connected drives. Locate the drive path corresponding to the flash drive, which will similar to `/dev/sdX`. 

112
2. Write the image to the flash drive, substituting the correct drive path.

    cd ~/nilmbuntu

    sudo dd if=nilmbuntu-16.04.2.iso of=/dev/sdX bs=1M

3. Run `sync` to ensure all drive operations have completed. Remove the drive and install onto a clean machine.

Testing in Virtual Machine

The installation image can be tested in a virtual machine to efficiently verify functionality before it is physically deployed.

1. Navigate to the nilmbuntu directory.

    cd ~/nilmbuntu

2. Create a virtual disk and boot up the graphical installer. This emulates booting from a Nilmbuntu flash drive. Run the installer to completion and then shut down the VM.

    ./run.sh -c

3. Boot from the virtual disk (without “flash drive” inserted) to test the installed Nilmbuntu system.

    ./run.sh -c

At this point, all the NILM Suite tools and configurations should be applied correctly. If not, the `standalone_install` should be fixed before re-building the image. Hot-fixes within the inner system are highly discouraged.
The disk created by the VM is located at `tmp-16.04.2/disk.img` and can be deleted to re-install the VM. Note that `cleanup.sh` destroys both the VM (test) and all the build files (inner system).

### A.2.5 Making Changes (Advanced)

**WARNING:** The following is intended only to outline general practices, not act as an instructional guide. All changes should be discussed with John and other developers.

#### Source Archive

Archived copies of the nilmbuntu and standalone_install repositories are available in `/archive` of the “Stick Maker” machine. The standalone installer contains all the dependent repositories of the NILM Suite and external dependencies. The archive should only be used in an emergency situation, when code has been compromised/lost.

```
  tar -xf /archive/nilmbuntu.tar.gz -C ~
  tar -xf /archive/standalone_install.tar.gz -C ~
```

#### Executable Scripts

Configurations and installations are performed in two places, each with a specific purpose.

System tools (ex. gparted, git, build-essentials) and configurations (ex. power settings) should occur within `customize-inner.sh`, since they pertain to the OS. These changes are applied once the inner system becomes interactive, called automatically at the end of `~/customize.sh`. 
Installation of NILM software occurs within the standalone_install. The `install.sh` file provides an overall installation flow and calls the individual installation scripts. Each program has a separate installer file located in `scripts/`. The `install-iso.sh` file acts mostly as a wrapper, since there are few actions that pertain to the NILM tools that are specific to the Nilmbuntu OS.

**Adding/Updating Repositories**

The `standalone_install` repository has git submodules for each of the individual NILM tools. The submodules should be initialized to a git tag \([43]\) rather than the HEAD of a branch. This ensures that version compatibility is preserved between modules.

A NILM repository (ex. nilm-capture) may have new commits (ex. bug fixes) that should be integrated in the next Nilmbuntu build. After committing and tagging these changes within a separate copy of the `nilmcapture` repository, navigate to the `nilmcapture` repository stored within the `standalone_install` repository. Pull and checkout the correct tag. Navigate back out to the `standalone_install` repository and commit the changes. This will only commit metadata indicating that the target commit of the submodule has changed; it will not commit duplicate code.
A.3 Nilm Management Console

A.3.1 Introduction

The NILM Management Console provides a graphical user interface for interacting with the NILM system to perform common administrative tasks and view relevant information in real-time. The console is organized into three tabs, described in detail below. The status tab allows the user to view both general and meter-specific error and warning messages, as well as information on CPU and disk use. From this tab, the user can also configure, calibrate, and run nilm-scope on meters. The drive tab allows the user to configure both primary and secondary storage. Finally, from the logs tab NILM log files can be viewed in real-time, and log and config files can be extracted to a destination of the user’s choosing. In addition to these tabs, the console also provides shortcuts for safely starting and stopping nilm-capture and the Apache database.

A.3.2 Managing capture and database

The console provides a safe way of starting and stopping nilm-capture and the Apache database. During normal operation, the database should always be started prior to the capture, and the capture should always be stopped prior to the database. The console enforces these constraints by periodically checking the state of the system and disallowing the user from completing certain actions accordingly. Possible states and their transitions are shown below.
The current state of the database and capture are displayed at the top of the console, next to the buttons for their management. They are updated every 0.5 seconds. Status Tab

A.3.3 Status Tab

The status tab provides an overview of the current state of the NILM system, updated every five seconds. It also provides shortcuts for running calibrate and nilm-scope, as well as for editing the meters.yml file.
Viewing overall system errors and information  In the status tab, the user can view overall system errors and warnings. If any overall problems are present, the indicator icon at the top of the box will turn red. Otherwise, it will be green. All warning and error messages are printed to the box, which is refreshed every five seconds along with the rest of the tab.

The status tab also provides the current CPU use of each core in the device, as
well as the current disk use of the drive mounted as primary data storage (the drive mounted to /opt/data). If any of these values are above 80%, a warning message will be generated.

<table>
<thead>
<tr>
<th>CPU use</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 1</td>
<td>17.0%</td>
</tr>
<tr>
<td>Core 2</td>
<td>19.19%</td>
</tr>
<tr>
<td>Disk use</td>
<td>22%</td>
</tr>
</tbody>
</table>

Figure A-11: Disk use and CPU usage

**Viewing meter-specific error messages** In addition to overall system errors, the status tab provides meter-specific error output. All meters either configured in the meters.yml file or detected by the NILM system as connected are shown on the right side of the status tab, along with a textbox for error output. If any errors specific to a particular meter are detected, the indicator next to that meter will turn red and an error message will be printed. Otherwise, the indicator will remain green.

![Figure A-12: Meter-specific error output](image)

**Running nilm-calibrate and nilm-scope** Buttons next to each meter provide shortcuts to run nilm-calibrate (for noncontact meters) and nilm-scope from the status tab. Both nilm-calibrate and nilm-scope can only be run when the database is running and the capture is stopped. To run nilm-calibrate for a noncontact meter, press the “Calibrate” button next to it’s name.
As shown below, this will open a new terminal running nilm-calibrate on the specified meter. After nilm-calibrate has finished running, the terminal must be closed in order to return to the management console.

To run nilm-scope for any meter, press the “Nilm Scope” button next to the name of the meter.
If there are no errors associated with the indicated meter, this will generate the dialog box shown below, allowing the user to choose which traces they would like to view for the meter—voltage, current, or both. Once a choice has been made, nilm-scope will run.

However, if there are errors associated with the meter, an error message such as the one shown below will be generated. Nilm-scope will not run if there is something wrong with the indicated meter.
**Configuring meters**  The “Configure Meters” button at the bottom right of the status tab provides a shortcut to safely edit the meters.yml file. It is only available when the database is running and the capture is stopped.

![Configure Meters](image)

Figure A-18: Configure Meters provides a shortcut to edit meters.yml

Pressing this button pulls up the meters.yml file in the Gedit text editor. After the user has finished their edits and closed the program, the new configuration is checked by running nilm-check-config. The user is prevented from saving the meters.yml file if errors are present. As shown below, if any errors are detected, the user is given the option of retrying their configuration or discarding their changes. This will continue until either the user has either created a configuration without errors or has chosen to restore the previous configuration.
A.3.4 Drive Tab

The drive tab is used to configure storage for the NILM database. It allows the user to view and manage all connected drives and their partitions, with the exception of the main hard drive. Each partition can be unmounted from its current mountpoint, formatted to the proper filesystem for the NILM database, and mounted either to `/opt/data` as primary storage or to `/opt/secondary` as secondary storage.
The drives currently mounted as primary and secondary are shown at the top of the tab, along with the space available in each.

Figure A-21: The drives currently mounted as primary and secondary are shown at the top of the drive tab

Storage may only be configured when both the database and capture have been stopped, as shown in the banner above. If the user attempts to perform any action while the database or capture are running, they will see the following error message.

Figure A-22: Storage cannot be configured until the database and capture are stopped

Mounting a drive as primary or secondary storage Once the database and capture have been stopped, storage may be configured. To mount a drive as primary or secondary, first unmount what is currently mounted there.
Then, unmount the desired drive from its current mountpoint. Press the unmount button next to the drive’s current mountpoint, and then choose “OK.”

If it is not already, the drive must then be formatted to the proper filesystem for the nilm database (ext4). Press the “Format” button for the desired drive and then choose “OK.”

Once the drive has been unmounted and properly formatted, it may be mounted either as primary or secondary storage.
Figure A-26: Disk sdb1 may now be used as primary or secondary storage

A.3.5 Logs Tab

The logs tab provides a graphical method of selecting and viewing the NILM log files, either statically or live. It also provides a shortcut to export the log and config files to a destination specified by the user.

Figure A-27: Overview of the Logs Tab

Viewing the logs All logs available to be viewed are listed as buttons on the left sidebar. The supervisor log and NilmDB intervals are always available, while the meters present will change depending on which meters are connected and configured. To view a log, select it from the sidebar.
Figure A-28: Selecting the Meter2 logs for viewing

By default, the logs are printed statically – in other words, the entire contents of the selected file are printed to the screen, and will not be updated until the button is pressed again. They can be navigated using the scrollbar. To view the logs live, the live checkbox must be selected, as shown below.
When the live checkbox is selected, the last (most recent) 100 lines of the selected file will be printed to the screen. They will be updated every 0.5 seconds for as long as the logs tab is still selected. When live is selected, the scrollbar is not available since the logs are constantly updating.
**Extracting** The extract button on the bottom right corner of the tab (shown below) copies the log (/var/log/nilm) and config (/opt/configs) folders into a single containing folder, converts it to a zip file, and saves this file to a location of the user’s choosing. Extract will always copy the entire contents of these folders, regardless of which file is currently being viewed in the tab.

![Figure A-30: Extracting log and config files](image)

When the extract button is pressed, the file dialog shown below will be brought up. The user may then edit the name of the zip file and choose a destination for it.
Figure A-31: Choosing a path location for extraction
Appendix B

Command-Line Reference

B.1 Indirect Calibration

Command: submeter

usage: submeter [-h] [-s] [-d] {cleanup, calibrate, capture, show} ...

# Upstream/Downstream Meter Calibration Tool
# Prerequisites:
- Capture must be STOPPED.
- NilmDB must be RUNNING.
- System clock must be +/- 60 seconds synchronized between machines.

Subcommands:
{cleanup, calibrate, capture, show}

mode of operation
cleanup cleanup residual files and NilmDB streams
calibrate calibrate sensors with existing input data
capture capture data from meters
show plot existing captured data

Options:
-h, --help show this help message and exit
-s, --save preserve temporary files and streams
-d, --debug verbose printouts of internal actions
Command: submeter capture

usage: submeter capture [-h] -m meter_name [-s raw_sensors] [-i file_iv] 
                      [-c prep_currents]

Capturing tool for acquiring raw samples and processing readings in 
preparation for the calibration algorithm.

Options:
    -h, --help       show this help message and exit
    -m meter_name    meter name (ex. meter1)
    -s raw_sensors   destination file for raw meter readings
    -i file_iv       destination file for processed IV data
    -c prep_currents destination file for current PREP data (single 
                      consolidated file)

Command: submeter show

usage: submeter show [-h] [-s file_sens] [-i file_iv] [-c file_prep] 
                    [-f file]

Plots data streams from existing capture files.

Options:
    -h, --help       show this help message and exit
    -s file_sens     raw non-contact sensor readings
    -i file_iv       processed IV data
    -c file_prep     current PREP data (single consolidated file)
    -f file          arbitrary timestamped data
Command: submeter calibrate

usage: submeter calibrate [-h] -m meter_name [-plots-off] [-p] [-r] [-e]
    -c prep_currents [prep_currents ...] -s
    raw_sensors [-o prep_sensors] [-i iv_sensors]
    [--param-tcc float] [--param-tbc float]
    [--param-tcs float] [--param-tbs float]
    [--plot-trans] [--plot-align] [--plot-diffs]

Determines non-contact meter calibration by matching loads between known
currents and unknown sensor readings.

Calibration results are automatically saved in the following path:

/opt/configs/submeter_cal

Options:
- `h`, --help        show this help message and exit
- `m` meter_name, --meter meter_name
  uncalibrated non-contact meter name (ex. meter1)
- `--plots-off`     do not generate plots (recommended for SSH sessions
  only)
- `-p`, --plots-hide do not show calibration plots upon completion
- `-r`, --reconstruct-only
  skip calibration, reconstruct raw sensor data into
  currents using existing calibration
- `-e`, --keep-vref  prevent auto-selecting voltage reference sensor,
  overwrites vref in meters.yml by default

Input Files:
- `c` prep_currents [prep_currents ...]
  calibrated meter’s current PREP streams - accepts
  single consolidated file (8xI cols) OR I individual
  files (8 cols) for I currents
- `-s` raw_sensors  uncalibrated non-contact meter’s raw sensor readings

Output Files:
  Save reconstructed data from newly calibrated non-contact meter.
- `-o` prep_sensors output PREP (consolidated format) of reconstructed
  currents
- `-i` iv_sensors  reconstructed IV

Adjustment Parameters:
- `--param-tcc` float  corr_thresh for current stream of transient detector
- `--param-tbc` float  bal_thresh for current stream of transient detector
- `--param-tcs` float  corr_thresh for sensor stream of transient detector
- `--param-tbs` float  bal_thresh for sensor stream of transient detector

Debugging:
- `--plot-trans`  plot transient detector intermediates and output
- `--plot-align`  plot correlation-based alignment intermediates
- `--plot-diffs`  plot aligned sensor/current transients
B.2 Metrics Generator

Command: metrics
usage: metrics [-h] [-A HOST_NILMDB] [-a HOST_MONGO] [-v]
{list,generate,extract,destroy,backup,restore} ...

Generate and manage metric data from load event streams.

Subcommands:
{list,generate,extract,destroy,backup,restore}
list list all loads
generate generate daily metrics from event streams
extract extract daily metrics for specified load
destroy daily daily metrics for specified load
backup backup database of daily metrics
restore restore database of daily metrics

Options:
-h, --help show this help message and exit
-A HOST_NILMDB, --host-nilmdb HOST_NILMDB
   NilmDB Server Address
-a HOST_MONGO, --host-mongo HOST_MONGO
   MongoDB Server Address
-v, --verbose verbose logging of internal actions

Command: metrics generate
usage: metrics generate [-h] [-f] [-c]

Generates summary metrics from NilmDB event streams. Without arguments,
calculates missing summary days until all metrics are calculated until
yesterday. Rolling metrics are also generated for the last 24 hours.
Configuration file is located at [metrics_config.json].

Options:
-h, --help show this help message and exit
-f, --force re-calculate and overwrite if metric exists for day
-c, --clean remove metrics for days not existing in event streams
Command: metrics extract

usage: metrics.py extract [-h] [-l load_id] [-s DATE_START] [-e DATE_END]

Extracts daily metrics for a specified load and date range. Results are printed to STDOUT by default, and they can be redirected to a JSON file.

optional arguments:
- -h, --help show this help message and exit
- -l load_id, --load load_id
  load identifier to extract
- -s DATE_START, --start DATE_START
  start date, inclusive (YYYY-MM-DD)
- -e DATE_END, --end DATE_END
  end date, inclusive (YYYY-MM-DD)

Command: metrics list

usage: metrics list [-h] [-m] [-e]

List all loads and metrics.

Options:
- -h, --help show this help message and exit
- -m, --metrics list set of metrics for each load
- -e, --extents show extent of time for each load (and metric)

Command: metrics destroy

usage: metrics generate [-h] [-f] [-c]

Generates summary metrics from NilmDB event streams. Without arguments, calculates missing summary days until all metrics are calculated until yesterday. Rolling metrics are also generated for the last 24 hours. Configuration file is located at [metrics_config.json].

Options:
- -h, --help show this help message and exit
- -f, --force re-calculate and overwrite if metric exists for day
- -c, --clean remove metrics for days not existing in event streams
Command: metrics backup


Backup all or part of metric database to file.

positional arguments:
    filepath             destination file

optional arguments:
    -h, --help           show this help message and exit
    -l load_id, --load load_id
                        load identifier to destroy
    -s DATE_START, --start DATE_START
                        start date, inclusive (YYYY-MM-DD)
    -e DATE_END, --end DATE_END
                        end date, inclusive (YYYY-MM-DD)

Command: metrics restore

usage: metrics restore [-h] [-f] filepath

Restores database of daily metrics from file. Conflicts are skipped by default.

positional arguments:
    filepath source file

optional arguments:
    -h, --help           show this help message and exit
    -f, --force          overwrites conflicting points
Appendix C

Source Code

C.1 Indirect Calibration

Source Code C.1

Filename: submeter.py

Collaborators: None

Description: Main entry point for indirect calibration. Handles command-line arguments from the user and controls the flow the program accordingly.

```python
#!/usr/bin/python

from shared import *
from data import *
from plot import *
from disaggregate import *
from commands import *
from helpers import *
from hardware import *

import sys, os, signal, shutil, glob
import numpy as np
import argparse

# /*============================================
# = Variables & Settings =
# ============================================*/

DATETIME = time_str()
```
DIR_TMP = "/tmp/submeter_%s" % DATETIME
NILMDB_TMP = "tmp_submeter_%s" % DATETIME
nilmdb_path = lambda stream_str: "/%s/%s" % (NILMDB_TMP, stream_str.strip().replace("-", "_"))
tmpfile_path = lambda file_str: os.path.normpath(os.path.join(DIR_TMP, "%s.dat" % file_str).replace("-", "_"))

DIR_SAVE = "/opt/configs/submeter_cal/cal_%s" % DATETIME
savedir_path = lambda file_str: os.path.normpath(os.path.join(DIR_SAVE, file_str))

# /*===================================
# = Subroutines =
# ===================================*/

def cleanup_active():
    if os.path.exists(DIR_TMP): # remove residual files
        if not os.path.isdir(DIR_TMP):
            os.remove(DIR_TMP)
        else:
            shutil.rmtree(DIR_TMP)
    nilmdb_destroy(nilmdb_path("*"))

def cleanup_residual():
    for f in glob.glob("/tmp/submeter_*"):
        if not os.path.isdir(f):
            os.remove(f)
        else:
            shutil.rmtree(f)
    nilmdb_destroy_tmp() # remove residual streams

def save_results(meter_name):
    if not os.path.exists(DIRSAVE):
        os.makedirs(DIRSAVE) # recursive parent creation
        # save all plots
        savePlots(savedir_path("plots.pdf"))
        # save configuration files
        copyFile(METERS_FILE, savedir_path("meters.yml"))
        calibration_path = os.path.join(CALIBRATION_DIR, "%s.yml" % meter_name)
        copyFile(calibration_path, savedir_path("%s.yml" % meter_name))

    # /*-------- Calibration Entry Points - depending on which data format provided --------*/

    # start with raw sensor data
def prep_sensors_from_raw(meter_name, sensor_path, autoselect=True):
        # override meters.yml with strongest vref index (optional) and scale
        scale_vref = autoselect_vref(args.meter_name, sensor_path, autoselect)
        sensors_prep_names = prep_sensors(meter_name, sensor_path)
        prep_paths = [tmpfile_path(name) for name in sensors_prep_names]
        return (prep_paths, scale_vref)
# /*---------- Tune Matrix & Rotations After Initial Determination
->----------*/

def sensors_process_tune(meter_name, file_raw, files_current_prep, matrix,
    rotations):
    # reconstruct IV->PREP raw sensors
    file_iv = tmpfile_path("%s-iv-tune" % meter_name)
    files_sensor_prep = [tmpfile_path("%s-prep-%i-tune" % (meter_name, i))
                        for i in xrange(num_phases(meter_name))]
    reconstruct_iv(meter_name, file_raw, file_iv)
    prep_currents(args.meter_name, file_iv, files_sensor_prep)

    data_verify = DataFiles(files_current_prep, files_sensor_prep,
            type_ids=("CURR", "CURR")) # re-purpose data class

    (matrix_offsets, rotation_offsets) = tune(data_verify, matrix, rotations)
    return (matrix_offsets, rotation_offsets)

# /*---------- Process Sensors Into IV & Display Verification Results
->----------*/

def sensors_process_verify(meter_name, file_raw, files_current_prep,
    plots_enabled):
    # reconstruct IV->PREP raw sensors
    file_iv = tmpfile_path("%s-iv" % meter_name)
    files_sensor_prep = [tmpfile_path("%s-prep-%i" % (meter_name, i))
                         for i in xrange(num_phases(meter_name))]
    reconstruct_iv(meter_name, file_raw, file_iv)
    prep_currents(args.meter_name, file_iv, files_sensor_prep)

    sensor_rotations = get_rotations(meter_name)

    data_verify = DataFiles(files_current_prep, files_sensor_prep,
            type_ids=("CURR", "CURR")) # re-purpose data class

    verify(data_verify, sensor_rotations, plots_enabled)
    return (file_iv, files_sensor_prep)

# /*========================================
# = Main Entry Point =
# ========================================*/
if __name__ == "__main__":
    # /*-------- CLI User Args --------*/
    # top level parser
term_rows, term_columns = map(int, os.popen('stty size', 'r').read().split())
hrule = "#" * term_columns
title = " Upstream/Downstream Meter Calibration Tool"
hrule = "#" * term_columns
parser = argparse.ArgumentParser(description="""%s
%s
%s

Prerequisites:
- Capture must be STOPPED.
- NilmDB must be RUNNING.
- System clock must be +/- 60 seconds synchronized between machines."""
hrule, title, hrule),
formatter_class=argparse.RawDescriptionHelpFormatter)
parser._positionals.title = "Subcommands"
parser._optionals.title = "Options"
parser.add_argument('-s', '--save', action='store_false',
dest="cleanup_temp",
help="preserve temporary files and streams")
parser.add_argument('-d', '--debug', action='store_true',
dest="debug_flag",
help="verbose printouts of internal actions")
subparsers = parser.add_subparsers(help="mode of operation")

# 'cleanup' mode parser
parser_cleanup = subparsers.add_parser("cleanup",
description="Cleanup residual files and NilmDB streams, left behind due to \nimproper exit or use of --save flag.",
help="cleanup residual files and NilmDB streams",
add_help=False)
parser_cleanup.set_defaults(mode="cleanup")

# 'calibrate' mode parser
parser_run = subparsers.add_parser("calibrate",
description="Determines non-contact meter calibration by matching loads between
known currents and unknown sensor readings.
Calibration results are automatically saved in the following path:
" /opt/configs/submeter_cal"),
help="calibrate sensors with existing input data",
formatter_class=argparse.RawDescriptionHelpFormatter)
parser_run.set_defaults(mode="calibrate")
parser_run._optionals.title = "Options"
parser_run.add_argument('-m', '--meter', type=str,
metavar='meter_name', dest="meter_name", required=True,
help="uncalibrated non-contact meter name (ex. meter1)"
parser_run.add_argument('--plots-off', action='store_false',
dest="plots_enabled",
help="do not generate plots (recommended for SSH sessions only)"
parser_run.add_argument('-p', '--plots-hide', action='store_false',
    dest='plots_show',
    help='do not show calibration plots upon completion')
parser_run.add_argument('-r', '--reconstruct-only', action='store_true',
    dest='skip_calibration',
    help='skip calibration, reconstruct raw sensor data into currents using existing calibration')
parser_run.add_argument('-e', '--keep-vref', action='store_false',
    dest='autosel_vref',
    help='prevent auto-selecting voltage reference sensor, overwrites vref in meters.yml by default')
parser_inputs = parser_run.add_argument_group("Input Files")
parser_inputs.add_argument('-c', nargs='+',
    metavar='prep_currents', dest='prep_currents', required=True,
    help='calibrated meter’s current PREP streams - accepts single consolidated file (8xI cols) OR I individual files (8 cols) for I currents')
parser_inputs.add_argument('-s', nargs=1,
    metavar='raw_sensors', dest='raw_sensors', required=True,
    help='uncalibrated non-contact meter’s raw sensor readings')
parser_outputs = parser_run.add_argument_group("Output Files",
    description="Save reconstructed data from newly calibrated non-contact meter.")
parser_outputs.add_argument('-o',
    metavar='prep_sensors', dest='prep_sensors', required=False,
    help='output PREP (consolidated format) of reconstructed currents')
parser_outputs.add_argument('-i',
    metavar='iv_sensors', dest='iv_sensors', required=False,
    help='reconstructed IV')
parser_params = parser_run.add_argument_group("Adjustment Parameters")
parser_params.add_argument('--param-tcc',
    metavar='float', dest='corr_thresh_curr', required=False, type=float,
    help='corr_thresh for current stream of transient detector')
parser_params.add_argument('--param-tbc',
    metavar='float', dest='bal_thresh_curr', required=False, type=float,
    help='bal_thresh for current stream of transient detector')
parser_params.add_argument('--param-tcs',
    metavar='float', dest='corr_thresh_sens', required=False, type=float,
    help='corr_thresh for sensor stream of transient detector')
parser_params.add_argument('--param-tbs',
    metavar='float', dest='bal_thresh_sens', required=False, type=float,
    help='bal_thresh for sensor stream of transient detector')
parser_debug = parser_run.add_argument_group("Debugging")
parser_debug.add_argument('--plot-trans',
    action='store_true', dest='plot_trans', required=False,
    help='plot transient detector intermediates and output')
parser_debug.add_argument('--plot-align',
    action='store_true', dest='plot_align', required=False,
    help='plot correlation-based alignment intermediates')
parser_debug.add_argument('--plot-diffs',
    action='store_true', dest='plot_diffs', required=False,
help="plot aligned sensor/current transients")

# 'capture' mode parser
parser_capture = subparsers.add_parser("capture",
   description="Capturing tool for acquiring raw samples and processing
   readings in preparation for the calibration algorithm.",
   help="capture data from meters")
parser_capture.set_defaults(mode="capture")
parser_capture._optionals.title = "Options"
parser_capture.add_argument("-m", type=str,
   metavar='meter_name', dest='meter_name', required=True,
   help="meter name (ex. meter1)")
parser_capture.add_argument("-s", nargs=1,
   metavar='raw_sensors', dest='meter_raw',
   help="destination file for raw meter readings")
parser_capture.add_argument("-i", nargs=1,
   metavar='file_iv', dest='meter_iv',
   help="destination file for processed IV data")
parser_capture.add_argument("-c", nargs=1,
   metavar='prep_currents', dest='meter_preps',
   help="destination file for current PREP data (single consolidated file)")

# 'show' mode parser
parser_show = subparsers.add_parser("show",
   description="Plots data streams from existing capture files.",
   help="plot existing captured data")
parser_show.set_defaults(mode="show")
parser_show._optionals.title = "Options"
parser_show.add_argument("-s",
   metavar='file_sens', dest='data_sens',
   help="raw non-contact sensor readings")
parser_show.add_argument("-i",
   metavar='file_iv', dest='data_iv',
   help="processed IV data")
parser_show.add_argument("-c",
   metavar='file_prep', dest='data_prep',
   help="current PREP data (single consolidated file)")
parser_show.add_argument("-f",
   metavar='file', dest='data_misc',
   help="arbitrary timestamped data")

# parse args from command line
if len(sys.argv) == 1:
   parser.print_help() # show help message by default
   sys.exit(1)
args = parser.parse_args()
if args.mode == "cleanup":
   cleanup_residual()
sys.exit(0)

# plot existing data files
elif args.mode == "show":
    # Raw Sensors
    if args.data_sens is not None:
        (ts, data) = load_data(args.data_sens)
        streams, titles = [], []
        for i in xrange(4):
            streams.append(data[:,i*2])
            titles.append("%i ($E_{%i}$)" % (i*2, i))
        for i in xrange(4):
            streams.append(data[:,i*2+1])
            titles.append("%i ($B_{%i}$)" % (i*2+1, i))
        plotGeneral("Raw Sensor Samples", streams, titles)
    # IV Data
    if args.data_iv is not None:
        (ts, data) = load_data(args.data_iv)
        streams, titles = [data[:,0]], ["V"]
        for i in xrange(1, data.shape[1]):
            streams.append(data[:,i])
            titles.append("$I_{%s}$" % chr(65+i-1))
        plotGeneral("Processed Current/Voltage Samples", streams, titles)
    # Current PREP
    if args.data_prep is not None:
        (ts, data) = load_data(args.data_prep)
        streams, titles = [], []
        for i in xrange(int(data.shape[1] / 8)):
            streams.append(data[:,(8*i):(8*i+2)])
            titles.append(chr(65+i))
        legend = ['P1', 'Q1']
        plotGeneral("Processed Current/Prep", streams, titles, legend)
    # Arbitrary Data
    if args.data_misc is not None:
        (ts, data) = load_data(args.data_misc)
        streams, titles = [], []
        for i in xrange(data.shape[1]):
            streams.append(data[:,i])
            titles.append("")
        plotGeneral("", streams, titles)
plt.show()
sys.exit(0)

else: # applicable for modes below
    os.mkdir(DIR_TMP) # create new temp directory
    (settings_meter, settings_calibration) = load_configs(args.meter_name) # ensures meter exists
    num_currents = num_phases(args.meter_name)
    def kill_handler(signal, frame):
        warn('Caught ^C. Cleaning temporary files/streams and exiting...')
cleanup_active()
sys.exit(1)
signal.signal(signal.SIGINT, kill_handler)

if args.mode == "calibrate":
    # set debug level and flags
    setDebug(args.debug_flag)
    setFlag('plot_trans', args.plot_trans)
    setFlag('plot_align', args.plot_align)
    setFlag('plot_diffs', args.plot_diffs)
    if args.corr_thresh_curr is not None: setFlag('corr_thresh_curr',
        args.corr_thresh_curr)
    if args.bal_thresh_curr is not None: setFlag('bal_thresh_curr',
        args.bal_thresh_curr)
    if args.corr_thresh_sens is not None: setFlag('corr_thresh_sens',
        args.corr_thresh_sens)
    if args.bal_thresh_sens is not None: setFlag('bal_thresh_sens',
        args.bal_thresh_sens)
    # current inputs
    if len(args.prep_currents) == 1: # consolidated PREP
        files_current_prep = parse_path(args.prep_currents[0])
    else: # individual PREP files
        files_current_prep = [parse_path(name) for name in args.prep_currents]
    if not args.skip_calibration:
        # sensor inputs
        (files_raw_prep, scale_vref) = prep_sensors_from_raw(args.meter_name,
            parse_path(args.raw_sensors[0]), args.autosel_vref)
        # delta system modifications
        has_neutral = meter_attr(args.meter_name, "calibration.has_neutral")
        # form data structure for known currents vs. uncalibrated sensors
        data = DataFiles(files_current_prep, files_raw_prep, type_ids=('CURR',
            "ADC_NC"), neutral_phase=None)
        if has_neutral: # standard WYE
            neutral_phases = (None, None)
        else: # DELTA - select one phase as 'neutral' to remove
            neutral_phases = select_neutral_phases(data)
        data.assign_neutral(neutral_phases[0]) # adjusts Data retroactively
        # run disaggregation and (optionally) plot results
        plot_grid = setupFitPlot(data) if args.plots_enabled else None
        transformation, sensor_rotations = disaggregate(data,
            plot_grid=plot_grid)
        # save transformation and rotations to calibration file
        (full_current_matrix, full_rotations) =
            generate_calibration_file(args.meter_name, scale_vref, transformation,
                sensor_rotations, neutral_phases[0])
        # refine delta matrix/rotation by calibrating with another neutral if
        # available (no derived phase)
        if not has_neutral and neutral_phases[1] is not None: # second
            neutral_available
            # data.unassign_neutral()
            # data.assign_neutral(neutral_phases[1])
# transformation, sensor_rotations = disaggregate(data,
plot_grid=None)
# (full_current_matrix_secondary, full_rotations_secondary) =
generate_calibration_file(args.meter_name, scale_vref, transformation,
sensor_rotations, neutral_phases[1])
# full_current_matrix[neutral_phases[0]] =
full_current_matrix_secondary[neutral_phases[0]] # substitute derived
row with secondary calibration
# full_rotations[neutral_phases[0]] =
full_rotations_secondary[neutral_phases[0]]
# generate_calibration_file(args.meter_name, scale_vref,
full_current_matrix, full_rotations, neutral_phase=None,
converted=True)
# # fine adjustment of resulting transformation and rotations
# (matrix_adj, rotations_adj) = sensors_process_tune(args.meter_name,
args.raw_sensors[0], files_current_prep, full_current_matrix,
full_rotations)
# generate_calibration_file(args.meter_name, scale_vref, matrix_adj,
rotations_adj, neutral_phase, converted=True)
# reconstruct currents from scratch using calculated matrix/rots
file_iv, files_sensor_prep = sensors_process_verify(args.
meter_name, args.raw_sensors[0], files_current_prep, args.
plots_enabled)
# (optional) save streams to files
if args.iv_sensors is not None:
  copyFile(file_iv, args.iv_sensors)
if args.prep_sensors is not None:
  consolidate_streams(files_sensor_prep, parse_path(args.
prep_sensors, False))
if not args.skip_calibration:
  # save plots and configurations
  save_results(args.meter_name)
  # plot results
if args.plots_enabled and args.plots_show: plt.show()
files_prep = [tmpfile_path("prep-%i"%i) for i in xrange(num_phases)]

# capture and process
capture_status = interactive_capture(args.meter_name, file_raw)
if capture_status != 0:
    os.remove(file_raw)  # remove captured file
    print "Cleaning temporary filesstreams and exiting..."
    cleanup_active()
    sys.exit(0)
if req_curr > 0:
    sys.stdout.write("Processing readings into currents...")
    reconstruct_iv(args.meter_name, file_raw, file_iv)
    sys.stdout.write("DONE" + os.linesep)
    num_iv_processed = int(execute("wc -l %s" % file_iv).split()[0])
    print "Processed: %i Samples" % num_iv_processed
if (args.meter_preps is not None):
    sys.stdout.write("Processing currents into PREP streams...")
    prep_currents(args.meter_name, file_iv, files_prep)
    consolidate_streams(files_prep, args.meter_preps[0])  # combine preps into one file
    sys.stdout.write("DONE" + os.linesep)
    num_prep_processed = map(lambda f: execute("wc -l %s" % f).split()[0], files_prep)
    print "Processed: [%s] Samples" % " ".join(num_prep_processed)
else:  # unknown mode
    sys.exit(1)

# /*---------- Clean-Up ----------*/
if args.cleanup_temp:
    cleanup_active()  # removes files and NilmDB streams used during calibration
from plot import *
from helpers import *

import numpy as np
import scipy.signal
from data import Data
import scipy.stats

# determines 'neutral' phase for delta connected system
# 'data' must have constructor arg 'neutral_phase=None'
# returns (i_best_neutral, i_second_best_neutral OR None)
def select_neutral_phases(data):
    SMUDGE_WIDTH = 10  # width=1 should theoretically disable smudging

    # load copies of diffs
    diff_phases, diff_sensors = data.diff_currents.copy(), data.diff_sensors.copy()

    # smudge sensor diffs to correct slight misalignment
    diff_sensors = smudge_stream(diff_sensors, SMUDGE_WIDTH)

    # stream of total overall sensor diff strength
    diff_sensor_strength = np.sum(np.abs(diff_sensors), axis=1)

    # eliminate multi-phase loads - remaining loads are to test neutral
    align_relative(diff_phases, width=(SMUDGE_WIDTH / 2))  # align diffs

    # count diffs between phases
    multi_count = np.zeros(diff_phases.shape[0])
    for ph_i in xrange(data.phases):
        multi_count[np.abs(diff_phases[:, ph_i]) != 0] += 1  # count diffs between phases
    diff_phases[multi_count != 2] = 0  # if not line-line load, zero all phases

    # determine total pickup on other phases
    matched, ph_letters = np.zeros(data.phases - 1), []
    for i in xrange(data.phases - 1):
        ph_i = i if i < i_neutral else i + 1  # adjusted phase index
        diff_ph = diff_phases[:, ph_i].copy()
        diff_ph[diff_sensor_strength == 0] = 0  # ignore unsensed loads
        matched[i] = np.sum(np.logical_and(diff_ph != 0,
                                           diff_phes[:, i_neutral] != 0))
ph_letters.append(data.phase_letter(ph_i))
if np.all(matched == 0):  # no current flows from other phases to i_neutral
    missing_loads = map(lambda x: "%s-%s" % (data.phase_letter(i_neutral), x), ph_letters)
    fail("Not enough information; need more sensed load transients on phase %s." % (" or ".join(missing_loads)))
favorability_factors[i_neutral] = np.amin(matched)  # maximizes number of diffs on each neutral phase
fav_max = np.amax(favorability_factors)
fav_idx = np.argsort(favorability_factors)  # worst to best
neutral_indices = [fav_idx[-1], fav_idx[-2]]
if favorability_factors[fav_idx[-2]] < 0.5 * fav_max:
    neutral_indices[1] = None  # no second-best
return tuple(neutral_indices)

# calculates single sensor/phase ratio in matrix
def factor(diff_ph, diff_s, plt):
    RATIO_MIN = 0.01  # property of sensor/distance
    if ((diff_ph != 0).sum() == 0):
        return 0  # no usable diffs
    fit_coeffs = np.linalg.lstsq(np.vstack([diff_ph, np.zeros(len(diff_ph))]).T, diff_s)[0]  # solve y=mx (through zero)
ratio = fit_coeffs[0]
ratio_initial = ratio
regr = fit_coeffs[0] * diff_ph
if plt is not None:
    plt.plot(diff_ph, regr, 'r-')  # initial regression line

    # remove outliers from regression before re-attempting fit -- outliers corresponds to coinciding ON/OFFs
    dist = np.abs(fit_coeffs[0] * diff_ph - diff_s) / (np.linalg.norm([fit_coeffs[0], -1]))
    nonzero_idx = (np.abs(dist) > 1)
dist_mean, dist_std = np.mean(dist[nonzero_idx]), np.std(dist[nonzero_idx])
    if dist_std > 0.1:  # re-do best fit line if all points not colinear
        valid_idx = (np.abs(dist - dist_mean) < 0.5 * dist_std)
        invalid_idx = np.invert(valid_idx)
        diff_ph[invalid_idx] = 0
        diff_s[invalid_idx] = 0
        if ((diff_ph != 0).sum() == 0):
            return 0  # no usable diffs
        fit_coeffs = np.linalg.lstsq(np.vstack([diff_ph, np.zeros(len(diff_ph))]).T, diff_s)[0]
```python
ratio = fit_coeffs[0]  # sensor / phase
regr = fit_coeffs[0]*diff_ph
if plt is not None:
    plt.plot(diff_ph, regr, 'g-')  # adjusted regression line

if (abs(ratio) < RATIO_MIN):
    return 0  # flat line - junk correlation
if plt is not None:
    plt.text(0.2*plt.gca().get_xlim()[1],0.8*plt.gca().get_ylim()[1],"%.3f" % ratio)

return ratio

# finds transformation matrix (currents to sensors) and phase rotations
def disaggregate(data, plot_grid=None):
    SMUDGE_WIDTH = 10  # width=1 should theoretically disable smudging
    MIN_ANGLE = 1  # min degrees sensor angle distance from axes before being invalid
    POINTS_WARN = 6  # min 'seen' load transients before warning user

    diff_phases, diff_sensors = data.diff_currents.copy(),
                               data.diff_sensors.copy()
    angle_sensors, angle_currents = data.angle_sensors.copy(),
                                  data.angle_currents.copy()

    # apply current diff signs to indicate transient direction (ON/OFF)
    for ph_i in xrange(data.phases):
        diff_phases[:,ph_i] = np.abs(diff_phases[:,ph_i]) *
                              np.sign(data.diff_preps_currents[ph_i][:,0])

    # smudge sensor diffs to correct slight misalignment - diff_phases dictates alignment
    diff_sensors, angle_sensors, angle_currents = smudge_stream(diff_sensors,
                                                               SMUDGE_WIDTH), smudge_stream(angle_sensors, SMUDGE_WIDTH),
                                               smudge_stream(angle_currents, SMUDGE_WIDTH)

    transformation = np.zeros((data.phases, data.sensors), dtype=np.float32)
    sensor_rotations = np.zeros(data.phases, dtype=np.float32)

    # eliminate multi-phase loads and those which are too close between phases
    if data.phases > 1:
        align_relative(diff_phases, width=(SMUDGE_WIDTH / 2))  # aligned diffs between phases
        multi_count = np.zeros(diff_phases.shape[0])  # count diffs between phases
        for ph_i in xrange(data.phases):
            multi_count[np.abs(diff_phases[:,ph_i]) != 0] += 1  # count diffs between phases
```

149
for ph_i in xrange(data.phases):
    diff_ph = diff_phases[:,ph_i]
    diff_ph[multi_count > 1] = 0

    # matches transients based on each pair
    diffs_pairs = []
    for ph_i in xrange(data.phases):
        stripped = []
        for s_i in xrange(data.sensors):
            diff_ph = diff_phases[:,ph_i].copy()  # copy, not reference
            diff_s = diff_sensors[:,s_i].copy()
            # eliminate if phase doesn’t appear on sensor
            idx_ex = np.where(np.logical_or(diff_ph == 0, diff_s == 0))[0]
            diff_s[idx_ex], diff_ph[idx_ex] = 0, 0
            # apply diff signs based on current transient directions (ON/OFF)
            diff_s *= np.sign(diff_ph)
            stripped.append({'diff_ph': diff_ph, 'diff_s': diff_s})
        diffs_pairs.append(stripped)

    # assign signs for sensors diffs based on angle changes
    master_sensors = list()
    for ph_i in xrange(data.phases):
        # determine master sensor for this phase
        i_max, val_max = 0, 0
        for s_i in xrange(data.sensors):
            pair = diffs_pairs[ph_i][s_i]
            diff_ph, diff_s = pair['diff_ph'], pair['diff_s']
            pair['ang_s'] = angle_sensors[:,s_i].copy()
            mask = (diff_ph != 0)
            mask_count = np.sum(np.abs(diff_s[mask]))
            if mask_count > val_max:
                i_max, val_max = s_i, mask_count
        master_sensors.append(i_max)

    master_s = diffs_pairs[ph_i][i_max]  # retrieve master sensor for this phase
    diff_mph, diff_ms, ang_ms = master_s['diff_ph'], master_s['diff_s'],
    master_s['ang_s']
    i_valid = np.where(diff_ms != 0)
    num_points = len(i_valid[0])
    if (num_points == 0):
        fail("No sensed load transients on Phase %s." %
        data.phase_letter(ph_i))
    if (num_points < POINTS_WARN):
        warn("Very few sensed load transients on Phase %s." %
        data.phase_letter(ph_i))

    # assign signs of each diff (based on angles relative to master)
    count_match, count_mismatch = 0, 0
    for s_i in xrange(data.sensors):
if i_max == s_i: continue  # skip master phase - defined as correct

this_s = diffs_pairs[ph_i][s_i]
diff_ph, diff_s, ang_s = this_s['diff_ph'], this_s['diff_s'],
this_s['ang_s']

# assign sign for each sensor diff relative to master sensor response
i_sens = np.where(diff_s != 0)
for i in i_sens[0]:
    if diff_ms[i] == 0:
        diff_s[i], diff_ph[i], ang_s[i] = 0, 0, 0  # eliminate anything the
    # master doesn't see
        continue

    ang_complement = angle_adjust(ang_ms[i] + 180)
closer_self = abs(angle_adjust(ang_ms[i] - ang_s[i]))
closer_complement = abs(angle_adjust(ang_complement - ang_s[i]))
diff_s[i] = np.abs(diff_s[i]) * np.sign(diff_ms[i])  # initially force
    # sign to master's
    if closer_complement < closer_self:  # closer to complement
        diff_s[i] *= -1  # switch to opposite sign

# compute transformation elements
for ph_i in xrange(data.phases):
    num_points = np.zeros(data.sensors)
    for s_i in xrange(data.sensors):
        diff_s = diffs_pairs[ph_i][s_i]['diff_s']
        num_points[s_i] = np.sum(diff_s != 0)
    std_points = np.std(num_points)

    for s_i in xrange(data.sensors):
        if num_points[s_i] < std_points: continue  # too few points - skip
            # sensor

        pair = diffs_pairs[ph_i][s_i]
        (diff_ph, diff_s) = pair['diff_ph'], pair['diff_s']

        plt = None
        if plot_grid is not None:
            (plt, grid) = plot_grid
            plt.figure(1)
            plt.plot(grid[ph_i, s_i])
        plt.plot(diff_ph[diff_s != 0], diff_s[diff_s != 0], 'b.')  # retained
            # data points

            transformation[ph_i, s_i] = factor(diff_ph, diff_s, plt)

    # determine sensor rotations for each phase
    for ph_i in xrange(data.phases):
        i_mast = master_sensors[ph_i]
        master_s = diffs_pairs[ph_i][i_mast]
        diff_mph, diff_ms, ang_ms = master_s['diff_ph'], master_s['diff_s'],
            master_s['ang_s']
ang_curr = angle_currents[:, ph_i]
mask_match = np.logical_and(diff_ms != 0, diff_mph != 0)
if (mask_match.sum() == 0):
    fail("No sensed load transients on Phase %s." %
data.phase_letter(ph_i))
mask_large = np.abs(diff_ms) > 0.2 *
    np.mean(np.abs(diff_ms[mask_match]))
mask_valid = np.logical_and(mask_match, mask_large)

# possibly, choose alternate sensor reference if angle too near axes
if data.sensors > 1:
    # if (abs(90 - ang_s_approx) < MIN_ANGLE):
    #     i_submast = np.argsort(np.abs(transformation[ph_i, :]))[-2] #
    #     # second highest pickup
    #     if np.abs(transformation[ph_i, i_submast]) > np.abs(0.5 *
    #         transformation[ph_i, i_mast]): # decent pickup on second sensor
    #         master_s = diffs_pairs[ph_i][i_submast]
    #         diff_mph, diff_ms, ang_ms = master_s['diff_ph'],
    #         master_s['diff_s'], master_s['ang_s']
    #         mask_match = np.logical_and(diff_ms != 0, diff_mph != 0)
    #         mask_large = np.logical_and(mask_match, mask_large)
    #         mask_valid = np.logical_and(mask_match, mask_large)
    #         if transformation[ph_i, i_submast] < 0: # complement of master
    #             ang_ms[mask_valid] += 180 # adjust afterwards
    rot_deltas = ang_ms[mask_valid] - ang_curr[mask_valid]
    for i in xrange(rot_deltas.shape[0]):
        rot_deltas[i] = angle_adjust(rot_deltas[i])
    rot_mode = circmedian(rot_deltas)
    sensor_rotations[ph_i] = np.deg2rad(rot_mode)

return transformation, sensor_rotations

def rotate(data):
    SMUDGE_WIDTH = 5 # width=1 should theoretically disable smudging
    # 'data' contains known/reconstructed currents
    preps_currents, preps_sensors = data.preps_currents, data.preps_sensors
diff_currents, diff_sensors = data.diff_currents, data.diff_sensors
angle_currents, angle_sensors = data.angle_currents, data.angle_sensors

# smudge sensor diffs to correct slight misalignment
diff_sensors, angle_sensors, angle_currents = smudge_stream(diff_sensors, SMUDGE_WIDTH),
smudge_stream(angle_sensors, SMUDGE_WIDTH),
smudge_stream(angle_currents, SMUDGE_WIDTH)

sensor_rotations = np.zeros(data.phases)
for ph_i in range(data.phases):
    ang_curr, ang_sens, diff_curr, diff_sens = angle_currents[:,ph_i],
    angle_sensors[:,ph_i], diff_currents[:,ph_i], diff_sensors[:,ph_i]

    mask_match = np.logical_and(diff_sens != 0, diff_curr != 0)
    mask_large = np.abs(diff_sens) > 0.2 *
    np.mean(np.abs(diff_sens[mask_match]))
    mask_valid = np.where(np.logical_and(mask_match, mask_large))

    rot_deltas = ang_sens - ang_curr
    rot_deltas = rot_deltas[mask_valid] # remove invalid points
    for i in range(rot_deltas.shape[0]): rot_deltas[i] =
    angle_adjust(rot_deltas[i])

    rot_mode = circmedian(rot_deltas)

    sensor_rotations[ph_i] = rot_mode

sensor_rotations = np.deg2rad(sensor_rotations)

return sensor_rotations

def tune(data, transformation, rotations):
    SMUDGE_WIDTH = 10 # width=1 should theoretically disable smudging
    MIN_POINTS = 4

    # adjust matrix - num_curr x num_sens
diff_currents = data.diff_currents
diff_sensors = data.diff_sensors
diff_sensors = smudge_stream(diff_sensors, SMUDGE_WIDTH)

curr_tune = np.zeros((data.phases, data.phases))
for i in range(data.phases):
    for j in range(data.phases):
        diff_curr, diff_sens = diff_currents[:,i], diff_sensors[:,j]
        mask_match = np.where(np.logical_and(diff_sens != 0, diff_curr != 0))

        if len(mask_match[0]) < MIN_POINTS: continue

        x, y = diff_sens[mask_match], diff_curr[mask_match]
        fit_coeffs = np.linalg.lstsq(np.vstack([x, np.zeros(len(x))]).T, y)[0]
        if i == j: # diagonal term - current scale
            curr_tune[i,j] = fit_coeffs[0] # ratio = correct/reconstructed
        else: # cross term - bleed
curr_tune[i,j] = -1.0/fit_coeffs[0]  # ratio = -(recon/correct)

# print curr_tune
matrix_adj = curr_tune.dot(transformation)

# adjust rotations - determine difference and add offset
rotation_offsets = rotate(data)
rotations_adj = rotations + rotation_offsets

return (matrix_adj, rotations_adj)

def verify(data_verify, sensor_rotations, plots_enabled=False):
    # rotation_offsets = rotate(data_verify)

    if plots_enabled:
        # plotConclusion(data_verify, sensor_rotations, rotation_offsets)
        plotCurrents(data_verify)
Filename: data.py

Collaborators: None

Description: Manages corresponding pair of current and sensor streams for a given calibration. Responsible for parsing data from input files. Aligns data streams using a correlation-based method.

```python
"""
Summary
"""

from helpers import *
from shared import *
from plot import plotDebug, plotShow, plotGeneral

import os
import calendar as cal
from datetime import datetime as dt
import numpy as np
import random
import scipy

# /*---------- Data Classes ----------*/

class Data(object):
    def __init__(self):
        (self.ts_currents, self.data_currents) = (None, None)
        (self.ts_sensors, self.data_sensors) = (None, None)

        @staticmethod
        def detect_rate(timestamps):
            dt = timestamps[-1] - timestamps[0]
            samples = timestamps.shape[0] - 1
            rate = 1.0e6 / (dt / float(samples))
            return (rate) # leave as float to fix minor skew

        @staticmethod
        def pad(timestamps, streams_list, width, mode='both'):
            ts_first, ts_last = timestamps[0], timestamps[-1]
            time_range = ts_last - ts_first
            samples = time_range.shape[0] - 1
            dt = (time_range / float(samples))

            ts_before = np.linspace(ts_first - width*dt, ts_first - dt, num=width)
            ts_after = np.linspace(ts_last + dt, ts_last + width*dt, num=width)

            if mode == 'both':
                ts_dst = np.hstack((ts_before, timestamps, ts_after))
            elif mode == 'before':
                ts_dst = np.hstack((ts_before, timestamps))
```
elif mode == 'after':
    ts_dst = np.hstack((timestamps, ts_after))
else:
    raise ValueError

streams_dst = list()
for stream in streams_list:
    zeros_before = stream[0, :]*np.ones((width, stream.shape[1]),
        dtype=stream.dtype)
    zeros_after = stream[-1, :]*np.ones((width, stream.shape[1]),
        dtype=stream.dtype)
    if mode == 'both':
        streams_dst.append(np.vstack((zeros_before, stream, zeros_after)))
    elif mode == 'before':
        streams_dst.append(np.vstack((zeros_before, stream)))
    elif mode == 'after':
        streams_dst.append(np.vstack((stream, zeros_after)))

assert(all([len(stream) == len(ts_dst) for stream in streams_dst]))
return (ts_dst, streams_dst)

# produces new ts/streams sampled at rate_dst, new streams different
# length
@staticmethod
def resample(rate_dst, ts_src, *streams_src):
    rate_src = Data.detect_rate(ts_src)
    ratio = float(rate_src) / float(rate_dst)  # float ftep to prevent
        roundoff
    length = int(ts_src.shape[0] / ratio)
    cols = streams_src[0].shape[1]
    streams_dst = [np.empty((length, cols), dtype=np.float32) for stream in
        streams_src]
    ts_dst = np.empty((length,), dtype=np.int64)
    for i in xrange(length):
        i_sample = int(round(ratio * i))  # nearest point in source
        i_sample = min(max(0, i_sample), ts_src.shape[0]-1)  # clamp to
        boundaries
        ts_dst[i] = ts_src[i_sample]
    for dst, src in zip(streams_dst, streams_src):
        dst[i] = src[i_sample]
    return (ts_dst, streams_dst)

# produces new ts/streams stretched/squeezed by offset_elements, new
# streams same length
@staticmethod
def resize(offset_elements, ts_src, *streams_src):
    ratio = float(len(ts_src)) / float(len(ts_src) + offset_elements)
length = ts_src.shape[0]  # same length
cols = streams_src[0].shape[1]

streams_dst = [np.empty((length, cols), dtype=np.float32) for stream in streams_src]
ts_dst = np.empty((length,), dtype=np.int64)

for i in xrange(length):
    i_sample = int(round(ratio * i))  # nearest point in source
    pad = (i_sample < 0 or i_sample >= length)
    ts_dst[i] = 0 if pad else ts_src[i_sample]
    for dst, src in zip(streams_dst, streams_src):
        dst[i] = 0 if pad else src[i_sample]

return (ts_dst, streams_dst)

# shifts data in place (pads, not rolls)
@staticmethod
def offset(data_stream, offset):
    t = -offset
    for c in xrange(data_stream.shape[1]):
        tmp = np.roll(data_stream[:,c], t)
        if t < 0:  # data leads (shift left)
            tmp[t:] = tmp[t-1]
        else:  # data lags (shift right)
            tmp[0:t] = tmp[t]
        data_stream[:,c] = tmp

class DataStreams(Data):
    def __init__(self, ts_currents, preps_currents, ts_sensors,
                 preps_sensors, type_ids, neutral_phase=None):
        ""
        Stores sensor readings and calibrated currents. Source is from
        pre-existing numpy arrays.
        ""
        self.transformation = None
        self.neutral_phase = neutral_phase
        self.type_ids = type_ids
self.hidden_neutral = None

self.load_streams(ts_currents, preps_currents, ts_sensors, preps_sensors)

# alignment of sensors/currents in stages
self.align_timestamps()
self.align_correlate()

# calculate derived streams (ex. diffs, magnitudes, angles)
self.calculate_streams()

if neutral_phase is not None:
    self.hide_neutral()

def load_streams(self, ts_currents, preps_currents, ts_sensors, preps_sensors):
    assert(all([arr.ndim == 1 for arr in [ts_currents, ts_sensors]]))  #
    # timestamps are 1D arrays
    assert(all([arr.ndim == 2 for arr in (preps_currents + preps_sensors)]))
    # data is 2D arrays
    assert(all([len(ts_currents) == len(arr) for arr in preps_currents]))  #
    # current preps match timestamp length
    assert(all([len(ts_sensors) == len(arr) for arr in preps_sensors]))  #
    # sensor preps match timestamp length

    self.ts_currents, self.preps_currents = ts_currents, preps_currents
    self.ts_sensors, self.preps_sensors = ts_sensors, preps_sensors

    self.phases = len(self.preps_currents)
    self.sensors = len(self.preps_sensors)

    # keep only P1/Q1
    for i in xrange(self.phases): self.preps_currents[i] =
        self.preps_currents[i][:,0:2]
    for i in xrange(self.sensors): self.preps_sensors[i] =
        self.preps_sensors[i][:,0:2]

    # align times between sources by timestamp
    # truncate streams and clamp length to shortest stream, leaving margin if
    # available
    def align_timestamps(self):
        MARGIN_POINTS = 3600
        def nearest_index(arr, val):
            return (np.abs(arr - val)).argmin()
        def margin_pad(ts, preps, start_idx, end_idx):
            # truncate with margins
            start_margin = max(start_idx - MARGIN_POINTS, 0)
            end_margin = min(end_idx + MARGIN_POINTS, len(ts))
            ts = ts[start_margin:end_margin]

    return (np.abs(arr - val)).argmin()
preps = [prep_stream[start_margin:end_margin] for prep_stream in preps]

# pad where margin insufficient (ran into boundary)
pad_before = MARGIN_POINTS - (start_idx - start_margin)
pad_after = MARGIN_POINTS - (end_margin - end_idx)
if pad_before != 0:
    ts, preps = Data.pad(ts, preps, pad_before, mode='before')
if pad_after != 0:
    ts, preps = Data.pad(ts, preps, pad_after, mode='after')

return (ts, preps)

# align beginning by timestamp
if self.ts_currents[0] < self.ts_sensors[0]: # currents begin before sensors - truncate currents
    c_start_idx = nearest_index(self.ts_currents, self.ts_sensors[0])
    s_start_idx = 0
else: # sensors begin before currents - truncate sensors
    c_start_idx = 0
    s_start_idx = nearest_index(self.ts_sensors, self.ts_currents[0])

# align ending by number of samples
(c_len, s_len) = len(self.ts_currents) - c_start_idx, len(self.ts_sensors)

if c_len < s_len: # currents end before sensors - truncate sensors
    c_end_idx = len(self.ts_currents)
    s_end_idx = s_start_idx + c_len
else: # sensors end before currents - truncate currents
    c_end_idx = c_start_idx + s_len
    s_end_idx = len(self.ts_sensors)

# apply margin/pad at beginning and end
self.ts_currents, self.preps_currents = margin_pad(self.ts_currents, self.preps_currents, c_start_idx, c_end_idx)
self.ts_sensors, self.preps_sensors = margin_pad(self.ts_sensors, self.preps_sensors, s_start_idx, s_end_idx)

assert(len(self.ts_currents) == len(self.ts_sensors))

def align_correlate(self):
    ALIGN_ABS_LIM = min(3600, (self.ts_currents.shape[0] - 60), (self.ts_sensors.shape[0] - 60)) # maximum allowable offset - 60 seconds

    mag_currents = median_filter(calc_magnitudes(*self.preps_currents), 7)
    mag_sensors = median_filter(calc_magnitudes(*self.preps_sensors), 7)
    sum_currents = np.sum((np.diff(mag_currents, axis=0)), axis=1)
    sum_sensors = np.sum((np.diff(mag_sensors, axis=0)), axis=1)
    sum_sensors = np.convolve(sum_sensors, np.ones(5), mode='same')
corr_max, corr_t = 0, 0
corr_vals = np.zeros(2*ALIGN_ABS_LIM+1)

for t in xrange(-ALIGN_ABS_LIM, ALIGN_ABS_LIM+1):
    if t >= 0:  # shift time left
        n = np.correlate(sum_currents[:t], sum_sensors[t:])[0]
    else:  # shift time right
        n = np.correlate(sum_currents[t:], sum_sensors[:t])[0]

corr_vals[t+ALIGN_ABS_LIM] = n
if n > corr_max:
    corr_max, corr_t = n, t

if getFlag('plot_align'):
    axes = plotGeneral(("Alignment Intermediates"),
        [mag_currents, mag_sensors, sum_currents, sum_sensors, corr_vals],
        ["Current\n Magnitudes", "Sensor\n Magnitudes", "Current\n Diff Sum",
         "Sensor\n Diff Sum", "Offset\n Correlation"],
        legend=None)
    axes[-1].text(0.95, 0.95,
        ("Sample Offset: %i" % (corr_t)),
        size='medium', va='top', ha='right', transform=axes[-1].transAxes)
    plotShow();

map((lambda sensor: Data.offset(sensor, corr_t)), self.preps_sensors)
printDebug("Alignment:", corr_t, "samples (+ is shift left)")

# hides neutral set in self.neutral_phase
# INTERNAL METHOD - USER should use 'assign_neutral'
def hide_neutral(self):
    assert(self.phases > 1)  # ensure system has enough poles
    assert(self.neutral_phase < self.phases)  # index in range
    assert(self.hidden_neutral is None)  # not already hidden

    hidden = dict()
    hidden["preps_currents"] = self.preps_currents.pop(self.neutral_phase)
    hidden["diff_preps_currents"] =
        self.diff_preps_currents.pop(self.neutral_phase)

    def pop(name):
        stream = getattr(self, name)
        hidden[name] = stream[:,self.neutral_phase]
        removed = np.delete(stream, self.neutral_phase, axis=1)
        setattr(self, name, removed)
        map(pop, ["diff_currents", "angle_currents")

    self.hidden_neutral = hidden
    self.phases -= 1

# restores neutral set in self.neutral_phase
# INTERNAL METHOD - USER should use 'unassign_neutral'
```python
def restore_neutral(self):
    assert(self.neutral_phase < self.phases + 1)  # index in range
    assert(self.hidden_neutral is not None)  # some must be hidden
    hidden = self.hidden_neutral
    self.preps_currents.insert(self.neutral_phase, hidden["preps_currents"])
    self.diff_preps_currents.insert(self.neutral_phase,
                                           hidden["diff_preps_currents"])

    def push(name):
        stream = getattr(self, name)
        stream = np.insert(stream, self.neutral_phase, hidden[name], axis=1)
        setattr(self, name, stream)
        map(push, ["diff_currents", "angle_currents"])

    self.hidden_neutral = None
    self.phases += 1

    # derived streams from P/Q data

def calculate_streams(self):
    # clean diffs - magnitude/angle of each P/Q change
    self.diff_preps_currents, self.diff_currents, self.angle_currents =
                              calc_diffs(self.preps_currents, self.type_ids[0])
    self.diff_preps_sensors, self.diff_sensors, self.angle_sensors =
                               calc_diffs(self.preps_sensors, self.type_ids[1])

    if getFlag('plot_trans'):
        plotShow()

    if getFlag('plot_diffs'):
        titles = [r'$\mathcal{D}(' + name + rnn + ')$' % (stream, index) for
                   (stream, index) in
                   zip(self.phases*['C'] + self.sensors*['S'],
                        map(self.phase_letter, range(self.phases)) +
                        range(self.sensors))]
        series = [prep[:, :, 2] for prep in
                   self.diff_preps_currents + self.diff_preps_sensors]
        axes = plotGeneral("Aligned Transients", series, titles,
                          legend=None)
        plotShow()

    # assign neutral phase to WYE and adjust data class if started with DELTA
    def assign_neutral(self, neutral_phase):
        if neutral_phase is None: return  # do nothing
        assert(self.neutral_phase is None)  # must begin with WYE Data
        self.neutral_phase = neutral_phase
        self.hide_neutral()

    # unassign neutral phase from DELTA and adjust data class if started with WYE
    def unassign_neutral(self):
        assert(self.neutral_phase is not None)  # must begin with DELTA Data
```

self.restore_neutral()
self.neutral_phase = None

# convert phase number to letter
def phase_letter(self, phase_idx):
    offset = int(self.neutral_phase is not None and phase_idx >= 0)
    return chr(65 + phase_idx + offset)

class DataFiles(DataStreams):
    def __init__(self, files_current_prep, files_sensor_prep, type_ids, neutral_phase=None):
        """Stores sensor readings and calibrated currents. Source is from space-separated text files."
        Arguments:
        files_current_prep {list(str) or FP} -- list of file paths to calibrated PREP, one for each phase
        OR file path to consolidated single file
        files_sensor_prep {list(str) or FP} -- list of file path to sensor reading PREP, one for each sensor
        OR file path to consolidated single file
        ""

        if type(files_current_prep) == list:  # list of indivudal PREP files
            timestamps_currents, preps_currents = zip(*map(load_data, files_current_prep))
            assert(all([timestamps_currents[0][0] == ts[0] for ts in timestamps_currents]))  # first timestamps match -- current preps
            assert(all([timestamps_currents[0][-1] == ts[-1] for ts in timestamps_currents]))  # last timestamps match -- current preps
            ts_currents = timestamps_currents[0]
            preps_currents = list(preps_currents)
        else:  # single consolidated PREP file
            ts_currents, stream_currents = load_data(files_current_prep)
            assert(stream_currents.shape[1] % 8 == 0)  # prep streams of the correct format
            preps_currents = []
            for i in xrange(stream_currents.shape[1] / 8):
                i_start, i_end = 8*i, 8*i+8
                preps_currents.append(stream_currents[:,i_start:i_end])

        if type(files_sensor_prep) == list:  # list of indivudal PREP files
            timestamps_sensors, preps_sensors = zip(*map(load_data, files_sensor_prep))
            assert(all([timestamps_sensors[0][0] == ts[0] for ts in timestamps_sensors]))  # first timestamps match -- sensor preps
assert(all([timestamps_sensors[0][-1] == ts[-1] for ts in timestamps_sensors])) # last timestamps match -- sensor preps
ts_sensors = timestamps_sensors[0]
preps_sensors = list(preps_sensors)
else: # single consolidated PREP file
ts_sensors, stream_sensors = load_data(files_sensor_prep)
assert(stream_sensors.shape[1] % 8 == 0) # prep streams of the correct format
preps_sensors = []
for i in xrange(stream_sensors.shape[1] // 8):
    i_start, i_end = 8*i, 8*i+8
    preps_sensors.append(stream_sensors[:,i_start:i_end])
super(DataFiles, self).__init__(ts_sensors, preps_sensors, type_ids, neutral_phase)

# /*---------- DEPRECATED CLASS BELOW ----------*/

class DataMockedSystem(Data):
    def __init__(self, phases, sensors, seconds, 
        loads_sub_single, loads_sub_multi, 
        loads_main_single, loads_main_multi, rate=3.0e3, 
        direction='downstream', transformation=None, 
        noise=None, time_offset=None):
        """Simulates main panel and subpanel of electrical. Generates sensor readings to emulate 'uncalibrated' panel.

        Arguments:
        phases {int} -- number of phases
        sensors {int} -- number of sensors ( > phases)
        seconds {long} -- seconds to simulate
        loads_sub_single {int} -- single-phase loads on subpanel
        loads_sub_multi {int} -- multi-phase loads on subpanel
        loads_main_single {int} -- additional single-phase loads on main panel
        loads_main_multi {int} -- additional multi-phase loads on main panel

        Keyword Arguments:
        rate {int} -- samples per second (default: {3.0e3})
        direction {str} --
            'upstream' generates sensor readings for main panel
            'downstream' for subpanel (default: {'downstream'})
        transformation {Array} --
            currents to sensors matrix, None for identity
        noise {float} -- percentage of noise to apply
            range is [0.0,1.0], None for no noise
        time_offset {int} -- samples to offset sensor samples/time
            positive offset leads (shift left), negative offset lags (shift right)
Data.__init__(self)
(self.WATTS_MIN, self.WATTS_MAX) = (500, 2000)
(self.MIN_DURATION, self.MAX_DURATION) = (.5, 5)

self.rate = rate
self.seconds = seconds
self.phases = phases
self.length = int(seconds*rate)
self.sensors = sensors
self.direction = direction
(self.loads_sub_single, self.loads_sub_multi) = (loads_sub_single, loads_sub_multi)
(self.loads_main_single, self.loads_main_multi) = (loads_main_single, loads_main_multi)
self.noise = noise
self.time_offset = 0 if time_offset is None else time_offset

self.data_sub, self.data_main, self.data_sensors = None, None, None
self.ts_sub, self.ts_main, self.ts_sensors = None, None, None

self.initTimestamps()
self.initTransformation(transformation)
self.initSubpanel()
self.initMainpanel()
self.initSensors()
self.applyNoise()
self.misalignData()

def initTimestamps(self):
    ts_begin = cal.timegm(dt.timetuple(dt.utcnow()))*1e6  # time now
    step = 1.0e6/self.rate
    ts = np.arange(0, 1.0e6*self.seconds, step, dtype=np.float32).astype(np.int64)
    timestamps = ts + ts_begin
    self.ts_sub, self.ts_main, self.ts_sensors = timestamps.copy(), timestamps.copy(), timestamps.copy()

def initTransformation(self, transformation):
    if transformation is not None:
        self.transformation = transformation
    return

    self.transformation = np.empty((self.phases, self.sensors),
                                   dtype=np.float32)
    for i in xrange(self.phases):
        dist = np.random.rand(1, self.sensors) - 0.5*np.ones((1, self.sensors))
        dist = dist / np.linalg.norm(dist)
        self.transformation[i, :] = dist
```python
def initSubpanel(self):
multiphase = self.mockPhase(self.loads_sub_multi)
self.data_sub = np.hstack([multiphase]*self.phases)
for i in xrange(self.phases):
    self.data_sub[:,[i]] += self.mockPhase(self.loads_sub_single)

def initMainpanel(self):
multiphase = self.mockPhase(self.loads_main_multi)
self.data_main = self.data_sub[:,:] + np.hstack([multiphase]*self.phases)
for i in xrange(self.phases):
    self.data_main[:,[i]] += self.mockPhase(self.loads_main_single)

def initSensors(self):
    if (self.direction not in ['upstream', 'downstream']) or 
       self.transformation.shape != (self.phases, self.sensors):
        raise ValueError
    if self.direction is 'upstream':  # main is uncalibrated
        self.data_sensors = self.data_main.dot(self.transformation)
    elif self.direction is 'downstream':  # sub is uncalibrated
        self.data_sensors = self.data_sub.dot(self.transformation)

def mockPhase(self, num_loads):
data = np.zeros([self.length, 1], dtype=np.float32)
for i in xrange(num_loads):
    random.seed(os.urandom(20))
    power = random.uniform(self.WATTS_MIN, self.WATTS_MAX)
    random.seed(os.urandom(20))
    duration = random.uniform(self.MIN_DURATION, self.MAX_DURATION)
    random.seed(os.urandom(20))
    time = random.uniform(0, self.seconds - duration - 0.1)
    (start, end) = (int(time*self.rate), int((time+duration)*self.rate))
    data[start:end] += power
return data

def applyNoise(self):
    def add_noise(data_raw):
        noise_width = 0
        for c in xrange(data_raw.shape[1]):
            width = self.noise * np.amax(np.abs(data_raw[:,c])) # proportional
            noise_width = max(noise_width, width)
            for c in xrange(data_raw.shape[1]):
                noise = np.random.normal(0, noise_width, self.length)
                data_raw[:,c] += noise
        if self.noise is not None:
            add_noise(self.data_sub)
            add_noise(self.data_main)
            add_noise(self.data_sensors)

def misalignData(self):
```

if self.time_offset is None or self.time_offset == 0:
    return

# adjust timestamps -- used for reference only
self.ts_sensors += self.time_offset * 1.0e6
if self.direction is 'upstream':  # main is uncalibrated
    self.ts_main += self.time_offset * 1.0e6
elif self.direction is 'downstream':  # sub is uncalibrated
    self.ts_sub += self.time_offset * 1.0e6

# pad data to cause misalignment
sample_offset = int(self.time_offset * self.rate)
Data.offset(self.data_sensors, sample_offset)
if self.direction is 'upstream':  # main is uncalibrated
    Data.offset(self.data_main, sample_offset)
elif self.direction is 'downstream':  # sub is uncalibrated
    Data.offset(self.data_sub, sample_offset)
from __future__ import print_function
import sys, os, shutil
import numpy as np
import scipy, scipy.signal, scipy.stats

from shared import *

def warn(warn_str):
    print("WARNING: " + str(warn_str), file=sys.stderr)

def fail(err_str):
    print("ERROR: " + str(err_str), file=sys.stderr)
sys.exit(1)

# DESC: converts PREP stream into magnitude (apparent power)
# USE: calc_magnitudes(prep_a, prep_b, prep_c)
# prep_X is a float32 numpy array with cols from nilm-prep, timestamps omitted
# RET: tuple of numpy arrays (ts, mags)
# mags is a 2D array, with result column for each PREP input
def calc_magnitudes(*prep_streams):
    # ensure synchronized prep streams
    assert(all(S.ndim == 2 for S in prep_streams))  # data is 2D arrays
    assert(all([len(prep_streams[0]) == len(arr) for arr in prep_streams]))  # current preps match lengths

    def calc_mag(P1, Q1):
        first_harmonic = np.hstack((P1[:,None], Q1[:,None]))
        return np.linalg.norm(first_harmonic, axis=1)

    def cat_results(streams):
        cols = map(lambda x: x[:,None], streams)
        return np.hstack(cols)

    mags = [calc_mag(S[:,0], S[:,1]) for S in prep_streams]
    return cat_results(mags)

# DESC: converts PREP stream into angles (Q/P arctan)
# USE: calc_angles(prep_a, prep_b, prep_c)
# prep_X is a flaot32 numpy array with cols from nilm-prep, timestamps omitted
# RET: tuple of numpy arrays (ts, angs)
# mag is a 2D array, with result column for each input
def calc_angles(*prep_streams):
    # ensure synchronized prep streams
    assert (all(S.ndim == 2 for S in prep_streams))  # data is 2D arrays
    assert (all([len(prep_streams[0]) == len(arr) for arr in prep_streams]))  # current preps match lenghs

def calc_angle(P1, Q1):
    invtan = np.arctan2(Q1, P1)
    return np.rad2deg(invtan)

def cat_results(streams):
    cols = map(lambda x: x[:, None], streams)
    return np.hstack(cols)

angs = [calc_angle(S[:, 0], S[:, 1]) for S in prep_streams]
return cat_results(angs)

def parse_path(filepath, check_exists=True):
    filepath = os.path.expanduser(filepath)
    if os.path.isabs(filepath):
        path = os.path.abspath(filepath)
    else:
        path = os.path.normpath(os.path.join(os.getcwd(), filepath))
    if check_exists and not os.path.exists(path):
        raise Exception("Input file/path does not exist.")
    return path

# DESC: loads text file into numpy arrays
# USE: load_data("~/Desktop/test/prep-a.dat")
# filepath accepts absolute, relative (to cwd), and home-relative path
# RET: tuple of numpy arrays (ts, data)
# ts is a 1D int64 array of timestamps
# data is a 2D float32 array, with dimensions matching the file’s columns
# NOTE: indistinguishable in speed from np.genfromtxt(...), but avoids truncation
def load_data(filepath):
    path = parse_path(filepath)  # ensure file exists
    f = open(path)
    lines = list()
    while True:
        l = f.readline().strip()
        if l == "":
            f.close()
break
lines.append(l.split())

rows, cols = len(lines), (len(lines[0]) - 1)
data_arr = np.empty((rows, cols), dtype=np.float32)
ts_arr = np.empty((rows,), dtype=np.int64)

for i in xrange(rows):
    l = lines[i]
    data_arr[i] = l[1:]  # automatic type casting
    ts_arr[i] = l[0]

return (ts_arr, data_arr)

def store_data(filepath, timestamps, *data_arrays):
    assert(all(len(timestamps) == len(arr) for arr in data_arrays))
    assert(timestamps.ndim == 1)  # timestamps are 1D arrays
    assert(all(S.ndim == 2 for S in data_arrays))  # data is 2D arrays

    filepath = parse_path(filepath, False)
    fout = open(filepath, "w")

    for i in xrange(len(timestamps)):
        cols = list()
        cols.append("%i" % timestamps[i])
        for data_arr in data_arrays:
            cols.extend(['{:.6e}'.format(data_arr[i,c]) for c in xrange(data_arr.shape[1])])
            fout.write(" " .join(cols) + os.linesep)

    fout.close()

def consolidate_streams(files_individual, file_consolidated):
    ts_streams, data_streams = zip(*map(load_data, files_individual))

    assert(all([ts_streams[0][0] == ts[0] for ts in ts_streams]))  # first timestamps match
    assert(all([ts_streams[0][-1] == ts[-1] for ts in ts_streams]))  # last timestamps match
    assert(all([len(ts_streams[0]) == len(ts) for ts in ts_streams]))  # same number of samples

    store_data(file_consolidated, ts_streams[0], *data_streams)

def copyFile(path_src, path_dst):
    shutil.copyfile(path_src, path_dst)

    # finds maximum voltage reference in sensor data (0-indexed)
def max_vrms(sensor_data):
        i_max, rms_max = (0, 0)
data = sensor_data[1].copy()
for i in xrange(0, data.shape[1], 2):  # even cols are voltages
    data[:,i] = np.mean(data[:,i])  # filter out DC
rms = np.sqrt(np.mean(np.square(data[:,i])))
if rms > rms_max:
    i_max, rms_max = i, rms
return i_max, rms_max
def angle_adjust(x):
    while x < -180: x += 360
    while x > 180: x -= 360
    return x
def median_filter(data_stream, width):
    return np.apply_along_axis(lambda n: scipy.signal.medfilt(n, kernel_size=width), axis=0, arr=data_stream)
def smudge_stream(stream, width):
    return np.apply_along_axis(lambda m: np.convolve(m, np.ones(width), mode='same'), axis=0, arr=stream)
    # algin diff cols relative to one another, assuming max shift width
def align_relative(diff_stream, width=5):
    if diff_stream.shape[1] == 1: return  # no need to align single col
    location_mask = np.sum(np.abs(diff_stream), axis=1)  # valid locations on all
    for c in xrange(diff_stream.shape[1]):
        diff_stream[:,c] = smudge_stream(diff_stream[:,c], (2*width+1))
    # remove duplicates, honor first occurence
    i, zero_until = 0, 0
    while i < diff_stream.shape[0]:
        if i < zero_until:
            diff_stream[i,c] = 0
        elif diff_stream[i,c] != 0:
            zero_until = i + width
        i += 1
col_diff_fn = lambda x: np.diff(x, axis=0)
def centered_std(data, width):
    mstd = np.zeros_like(data)
    # centered rolling std with odd width
    assert(width % 2 == 1)
    radius = width/2
    for i in xrange(radius, len(data) - radius):
        mstd[i] = np.std(data[i-radius:i+radius+1])
    return mstd
def trailing_std(data, width):
mstd = np.zeros_like(data)
for i in xrange(width-1, len(data)):
    mstd[i] = np.std(data[i-width+1:i+1])
return mstd

def leading_std(data, width):
    mstd = np.zeros_like(data)
    for i in xrange(len(data)-width):
        mstd[i] = np.std(data[i:i+width])
    return mstd

# main diff handler

def diff(data_stream, type_id, common_detect=False):
    flag_default = lambda flag, default: getFlag(flag) if hasFlag(flag) else default
    if type_id == "CURR":
        corr_thresh, balance_thresh = flag_default('corr_thresh_curr', 10.0),
        flag_default('bal_thresh_curr', 10.0)
    elif type_id == "ADC_NC":
        corr_thresh, balance_thresh = flag_default('corr_thresh_sens', 50.0),
        flag_default('bal_thresh_sens', 10.0)
    else:
        fail("Diff of unrecognized stream type.")
    return diffs_movstd(data_stream, corr_thresh, balance_thresh, common_detect)

def label_type(type_id):
    label = ""
    if type_id == "CURR":
        label = "Current"
    elif type_id == "ADC_NC":
        label = "Sensor"
    return label

def smart_diff(data_stream, diff_fn=col_diff_fn, sensitivity=(1,1),
                common_threshold=False):
    stream = data_stream[1:,] # skip first to align to diffs
    diff = diff_fn(data_stream)
    smart_diffs = np.zeros_like(diff)

    # determine noise thresholds
    sensitivity_diff, sensitivity_noise = sensitivity
    diffs_noise = np.apply_along_axis(lambda m: scipy.signal.medfilt(m,
                                      kernel_size=3), axis=0, arr=diffs) # eliminate peaks
    std_diffs = np.ones(diffs.shape[1])*np.std(diffs) if common_threshold else
                np.std(diffs, axis=0)
    thresh_diff = sensitivity_diff * std_diffs # hysteresis between detecting
                      transient and ascertaining SS
    thresh_noise = sensitivity_noise * std_diffs
for c in xrange(diffs.shape[1]):
    ss_val, ss_idx, diff_idx = 0, 0, -1  # last steady state
    swinging = False  # currently undergoing change

    i = -1
    while i <= len(diffs):
        i += 1
        delta = abs(diffs[i,c]) if i < len(diffs) else 0

        if not swinging and (delta > thresh_diff[c] or i >= len(diffs)):
            # start rising/falling OR end of stream
            # resolve/ignore previous diff
            if i < ss_idx + 20:  # ensure diffs are not too close together
                diff_idx = -1  # effectively ignores prev and curr diff -
                unstable/short ss level
                continue
            avg = np.mean(stream[ss_idx+10:i-2,c])
            if diff_idx != -1:
                smart_diffs[diff_idx,c] = (avg - ss_val)
                # remember for this diff
                diff_idx = i  # mark beginning of diff
                ss_val = avg
                swinging = True
                i += 10  # skip samples while stabilizing (until strictly incr/decr)

        if swinging and delta < thresh_noise[c]:  # finish rising/falling
            swinging = False
            ss_idx = i  # begin next ss interval

        else:  # not swinging and small change OR still swinging and large
            # change
            pass

    return smart_diffs

def diffs_movstd(data_stream, corr_thresh, balance_thresh,
                 common_detect=False):
    """Finds clean diffs within a data stream.
Arguments:
    data_stream {Array2D} -- time along axis 0, streams along axis 1 (shape
    N x C)
    corr_thresh {float} -- correlation diff detection threshold
    balance_thresh {float} -- maximum allowable difference between
    before/after std
    common_detect {bool} -- detect along magnitude of axis 1 (default:
    False)"""
Returns:

`Array2D -- clean diff stream (shape N-1 x C)`

```
MEDFILT_WIDTH  = 7  # pre-smoothing median filter
MEASURE_AFTER  = 50  # useful sample width after diff
SAFE_AFTER     = 25   # ignore sample width after diff
MEASURE_BEFORE = 30  # useful sample width before diff
SAFE_BEFORE    = 5   # ignore sample width before diff

def detect(data):
    # zero crossings of sinusoids in delta_std are good diffs
    delta_std = trailing_std(data, MEASURE_AFTER) - leading_std(data, MEASURE_BEFORE)
    delta_std = delta_std[1:]  # typical diff stream
    # generate step exemplar
    width_step = max(MEASURE_AFTER, MEASURE_BEFORE)
    step = np.hstack((np.zeros(width_step), np.ones(width_step)))
    exemplar = trailing_std(step, MEASURE_AFTER) - leading_std(step, MEASURE_BEFORE)

    # locate all neg->pos transitions at each good diff
    corr = np.correlate(delta_std, exemplar, mode="same")
    corr[corr < 0] = 0
    corr_debug = corr.copy()
    corr[corr < corr_thresh] = 0  # determined empirically
    idx_peaks = scipy.signal.argrelextrema(corr, np.greater)[0]

    # ensure diffs are far enough apart
    diff_indices = []
    for i, idx in enumerate(idx_peaks):
        # ensure no diffs are too close
        idx_prev = idx_peaks[i-1] if i > 0 else 0
        if idx - idx_prev <= MEASURE_BEFORE: continue  # prev diff too close
        idx_next = idx_peaks[i+1] if i+1 < len(idx_peaks) else len(data)
        if idx_next - idx <= MEASURE_BEFORE: continue  # next diff too close
        if idx < MEASURE_BEFORE or idx + MEASURE_AFTER >= len(data): continue  # boundary check

        diff_indices.append(idx)

    return diff_indices, corr_debug

def measure(diff_indices, data):
    diffs_vals = np.zeros_like(data)
    used_indices = []
    for idx_diff in diff_indices:
        # average SS on either side of diff (between SAFE_GAP and MEASURE_WIDTH) to measure delta
        data_before = data[(idx_diff - MEASURE_BEFORE):(idx_diff - SAFE_BEFORE)]
        ss_before, std_before = np.mean(data_before), np.std(data_before)
data_after = data[(idx_diff + SAFE_AFTER + 1): (idx_diff + MEASURE_AFTER + 1)]
ss_after, std_after = np.mean(data_after), np.std(data_after)
if abs(std_before - std_after) > balance_thresh: continue  # SS

diffs_vals[idx_diff] = diff_value
used_indices.append(idx_diff)

return used_indices, diffs_vals

def balance_stream(data):
    bal = np.zeros_like(data)
    for i in xrange(MEASURE_BEFORE, bal.shape[0] - MEASURE_AFTER):
        data_before = data[(i - MEASURE_BEFORE): (i - SAFE_BEFORE)]
        std_before = np.std(data_before)
        data_after = data[(i + SAFE_AFTER + 1): (i + MEASURE_AFTER + 1)]
        std_after = np.std(data_after)
        bal[i] = abs(std_before - std_after)
    return bal

def detect_measure(data):
    diff_locs, corr_debug = detect(data)
    used_indices, diffs_vals = measure(diff_locs, data)
    return diffs_vals

data_smoothed = median_filter(data_stream, MEDFILT_WIDTH)

if common_detect:
    data_detect = np.linalg.norm(data_smoothed, axis=1)
    diff_locs, corr_debug = detect(data_detect)
    valid_mask = np.zeros(len(data_detect), dtype=bool)
    valid_mask[diff_locs] = True
    def common_measure(col):
        used_indices, diffs_vals = measure(diff_locs, col)
        invalid_remove = list(set(diff_locs) - set(used_indices))
        valid_mask[invalid_remove] = False
        return diffs_vals
    diff_stream = np.apply_along_axis(lambda c: common_measure(c)), axis=0, arr=data_smoothed)
    diff_stream[valid_mask == 0] = 0
    if getFlag('plot_trans'):
        bal_debug = np.apply_along_axis(lambda c: balance_stream(c)), axis=0, arr=data_smoothed)
        label = (' - %s' % getFlag("diff_label") if hasFlag("diff_label")
        else "")
    from plot import plotGeneral
    axes = plotGeneral("%s %s" % label),
    [data_stream, data_detect, corr_debug, bal_debug, diff_stream],
    ["Smoothed\nInput\n\nData", "Magnitude", "Volatility\n\nCorrelation", "Balance", "Transient\nStream"])
legend=None)
axes[-1].text(0.95, 0.95,
"corr_thres: %.2f\nbalance_thres: %.2f" % (corr_thres,
balance_thres),
size='medium', va='top', ha='right', transform=axes[-1].transAxes)
else:
diff_stream = np.apply_along_axis(detect_measure, axis=0,
arr=data_smoothed)
return diff_stream
def integrate_diffs(diffs):
recon = np.zeros((diffs.shape[0] + 1, diffs.shape[1]))
for col in xrange(diffs.shape[1]):
    for i in xrange(len(diffs)):
        if diffs[i, col] != 0:  # emulate short transient
            recon[i+1:i+11, col] = diffs[i,col]
return recon
def calc_diffs(preps_streams, type_id):
diffs_prep = []
for i, stream in enumerate(preps_streams):
    phase = chr(65+i) if type_id == 'CURR' else str(i)
    setFlag("diff_label", "%s %s" % (label_type(type_id), phase))
    diffs_prep.append(diff(stream[:, :, 2], type_id, common_detect=True))

diff_angles = calc_angles(*diffs_prep)  # angle formed by Q1/P1 change ONLY
diff_mags = calc_magnitudes(*diffs_prep)  # magnitude of P/Q change ONLY
diff_angles[diff_mags == 0] = 0
return (diffs_prep, diff_mags, diff_angles)

# SOURCE: https://github.com/scipy/scipy/issues/6644
def circmedian(angs):
    angs = np.deg2rad(angs)
    pdists = angs[np.newaxis, :] - angs[:, np.newaxis]
    pdists = (pdists + np.pi) % (2 * np.pi) - np.pi
    pdists = np.abs(pdists).sum(1)
    return np.rad2deg(angs[np.argmin(pdists)])

# determine last phase row/rotation for delta system
# mostly adapted from calibration code 'calibrate_finalize.py'
# current transformation is P x S, return (P + 1) x S
# NOTE: delta phase rotation intrinsically applied through submeter cal
def complete_transformation(current_matrix, sinefit_rotations, neutral_phase):
    if neutral_phase is None:  # already full versions
        return (current_matrix, sinefit_rotations)
    (P, S) = current_matrix.shape
    assert(P == 2)  # only supports 3-P delta system (2P delta == 1P + N)
# check direction - flip sense from CA to AC when 'neutral' is C
phase_angle = (sinefit_rotations[0] - sinefit_rotations[1]) % (2*np.pi)
if(np.cos(phase_angle) > 0): # 60/300 deg is AC/CB -> convert to AC/BC
current_matrix[1] *= -1

# determine last phase rotation
phase_angle = (sinefit_rotations[0] - sinefit_rotations[1]) % (2*np.pi)
if(np.sin(phase_angle) < 0): # rot 1 leads by 120
    last_rot = sinefit_rotations[1] + (2*np.pi/3.0)
else: # rot 0 leads by 120
    last_rot = sinefit_rotations[0] + (2*np.pi/3.0)

sinefit_rotations %= (2*np.pi)

# full sinefit rotations
full_rotations = np.insert(sinefit_rotations, neutral_phase, last_rot)

# compute full current matrix
conversion_matrix = np.insert(np.eye(P), neutral_phase, -np.ones(P), axis=0)
full_current_matrix = conversion_matrix.dot(current_matrix)

return full_current_matrix, full_rotations
import usb_sensor
import ethernet_sensor
from reconstructor import Reconstructor
from commands import *
from submeter import nilmdb_path, tmpfile_path
import yaml
import re
import sys, os, shutil
import numpy as np
import time
import readline
from threading import Thread
from serial.serialutil import SerialException

""
Data capturing tools. Adapted from nilm-capture.
""

# /*---------- Vars & Configs ----------*/
NON_CONTACT_NUM_LINES=3000
CONTACT_NUM_LINES = 6000*3
NP_BUFFER_SIZE = int(CONTACT_NUM_LINES*2)
NON_CONTACT_RATE, CONTACT_RATE = 3000, 6000
METERS_FILE = "/opt/configs/meters.yml"
CALIBRATION_DIR = "/opt/configs/meters"
METER_DEV = "/dev/nilm/%s-data"
TYPE_NON_CONTACT = "noncontact"
TYPE_CONTACT = "contact"

# /*---------- Helpers ----------*/
# generate contact meter calibration file (first use)
def setup_contact_calibration(meter_name, settings_meter):
    calibration_path = CALIBRATION_DIR+"/"+meter_name+".yml"
#build the recovery matrix

#This is just a scaled permutation matrix

sensor_config = settings_meter["sensors"]
current_indices = sensor_config["current"]["sensor_indices"]
current_scales = sensor_config["current"]["sensor_scales"]
voltage_indices = sensor_config["voltage"]["sensor_indices"]
voltage_scales = sensor_config["voltage"]["sensor_scales"]

# if this is a scalar, convert it to a list
if (not(type(current_scales) is list)):
current_scales = np.ones(settings_meter["phases"])*current_scales
if (not(type(voltage_scales) is list)):
voltage_scales = np.ones(settings_meter["phases"])*voltage_scales

# The iv matrix holds voltages in the first N rows and currents in the next N rows.
# Multiplying the raw data by this matrix produces the IV data rows directly
iv_matrix = np.zeros((settings_meter["phases"]*2,6))
for i in range(settings_meter["phases"]):
x = np.zeros(6)
x[voltage_indices[i]] = voltage_scales[i]
iv_matrix[i] = x

for i in range(settings_meter["phases"]):
x = np.zeros(6)
x[current_indices[i]] = current_scales[i]
iv_matrix[i + settings_meter["phases"]] = x

#now write out the prep rotations
sinefit_rotations =
    settings_meter["sensors"]["current"]["sinefit_rotations"]

#convert rotations to radians
sinefit_rotations = [x*(2*np.pi)/360.0 for x in sinefit_rotations]

with open(calibration_path,'w') as f:
    config = {
        "iv_matrix": iv_matrix.tolist(),
        "sinefit_rotations": sinefit_rotations
    }
    f.write(yaml.safe_dump(config, default_flow_style=False,
        canonical=False))

def load_configs(meter_name):
    try:
        with open(METERS_FILE,'r') as f:
            settings_configs = yaml.load(f)
    except IOError:
        fail("Can’t load meters file at [%s]." % METERS_FILE)
    except KeyError as e:
        fail("Can’t find [%s] in meters.yml file." % meter_name)
    try:
        settings_meter = settings_configs[meter_name]
    except KeyError as e:
        fail("Can’t find [%s] in meters.yml file." % meter_name)
    try:
        with open(CALIBRATION_DIR+"/"+meter_name+".yml","r") as f:
            settings_calibration = yaml.load(f)
except IOError:
    settings_calibration = None

return (settings_meter, settings_calibration)

# Reads meters file (or calibration file) and retrieves attribute
# ex. meter_attr("meter1", "sensors.current.sensor_indices", False)
#     -> settings_meter["sensors"]["current"]["sensor_indices"]
# ex. meter_attr("meter1", "sinefit_rotations", True)
#     -> settings_calibration["sinefit_rotations"]
def meter_attr(meter_name, attr_hierarchy, from_calibration=False):
    (settings_meter, settings_calibration) = load_configs(meter_name)
    settings = settings_calibration if from_calibration else settings_meter
    attr_steps = attr_hierarchy.split(".")
    try:
        while len(attr_steps) > 0:
            settings = settings[attr_steps.pop(0)]
    except KeyError as e:
        fail("Error reading config for [%s], missing [%s]." % (meter_name, e[0]))
    return settings

def num_phases(meter_name):
    return meter_attr(meter_name, "phases")

def get_rotations(meter_name):
    has_neutral = meter_attr(meter_name, "calibration.has_neutral")
    rot_key = 'sinefit_rotations' if has_neutral else 'full_sinefit_rotations'
    return meter_attr(meter_name, rot_key, from_calibration=True)

def check_calibration(meter_name):
    (settings_meter, settings_calibration) = load_configs(meter_name) # settings_meter is valid
    if settings_meter["type"] == TYPE_NON_CONTACT and settings_calibration is None:
        return False
    return True # calibrated NC or valid Contact

# /*---------- Capture Raw and Reconstruct IV ----------*/
class Capture:
    def __init__(self, meter_name, settings_meter, file_raw):
        self.meter_name, self.settings_meter, self.file_raw = meter_name,
            settings_meter, file_raw
        self.running = False
        self.samples_captured = 0
        self.rate = 1
        self.thread = None

    def start(self):
        if self.thread is None:
self.running = True
self.samples_captured = 0
self.thread = Thread(target=self.capture_raw)
self.thread.start()

def stop(self):
    if self.thread is not None:
        self.running = False
        self.thread.join()
        self.thread = None

def num_samples(self):
    return self.samples_captured

def num_seconds(self):
    return int(round(self.samples_captured / self.rate))

def capture_raw(self):
    if not self.running: return
    settings_meter, file_raw = self.settings_meter, self.file_raw
    try:
        if(settings_meter["enabled"] == False):
            print "[\%s] not enabled, exiting capture" % self.meter_name
            sys.exit(0)

        num_phases = settings_meter["phases"]
        meter_type = settings_meter["type"]
        if(meter_type == TYPE_NON_CONTACT):
            meter_serial = settings_meter["serial_number"]
            self.rate = NON_CONTACT_RATE
        else:
            meter_ip_addr = settings_meter["ip_address"]
            self.rate = CONTACT_RATE

        except KeyError as e:
            fail("Error reading config for [\%s], missing [\%s]." %
            (self.meter_name, e[0]))

        if(meter_type == TYPE_NON_CONTACT):
            try:
                meter = usb_sensor.Sensor(METER_DEV % meter_serial,
                NON_CONTACT_NUM_LINES)
            except SerialException as e:
                fail("Cannot find USB meter [\%s] with serial number [\%s]." %
                (self.meter_name, meter_serial))
            else:
                meter = ethernet_sensor.Sensor(meter_ip_addr, CONTACT_NUM_LINES)

        try:
            fout = open(file_raw, "w")
        except IOError:
fail("Could not open file [%s] for writing." % file_raw)

dc_offset = None
while self.running:
    # read sample batch from sensors
    try:
        data = meter.read_ser()
    except IOError:
        warn("Signal interrupted adc read\n"
            continue
    if(data is None):
        warn("Port closed, exiting capture\n")
        self.running = False
        raise IOError

#remove the DC offset
if(dc_offset is None):
    dc_offset = np.average(data[:,1:],0) # first batch only
    data[:,1:] -= dc_offset

# write batch to file
try:
    for i in xrange(len(data)):
        cols = ["%i" % data[i,c] for c in xrange(data.shape[1])]
        fout.write(" ".join(cols) + os.linesep)
    self.samples_captured += len(data)
except IOError:
    fail("Could not write to file [%s]." % file_raw)

meter.close_ser()
fout.close() # safely end writing

def reconstruct_iv(meter_name, file_raw, file_iv):
    (settings_meter, settings_calibration) = load_configs(meter_name)
    try:
        if(settings_meter["enabled"]==False):
            warn("[s] not enabled, exiting capture% meter_name)
            sys.exit(0)

        # init buffers
        meter_type = settings_meter["type"]
        if(meter_type==TYPE_NON_CONTACT):
            raw_buffer = np.zeros((NP_BUFFER_SIZE, 9), dtype=np.float64)
            iv_buffer = np.zeros((NP_BUFFER_SIZE, settings_meter['phases']+2),
                dtype=np.float64)
        elif(meter_type==TYPE_CONTACT):
            setup_contact_calibration(meter_name, settings_meter)
            raw_buffer = np.zeros((NP_BUFFER_SIZE, 7), dtype=np.float64)
            iv_buffer = np.zeros((NP_BUFFER_SIZE, settings_meter['phases'])*2+1),
                dtype=np.float64)
else:
    fail("Invalid meter type.")

eexcept KeyError as e:
    fail("Error reading config for [%s], missing [%s]." % (meter_name,e[0]))

reconstructor = Reconstructor(meter_name, settings_meter)

# init I/O
try:
    fin = open(file_raw, "r")
    fout = open(file_iv, "w")
except IOError as e:
    fail("Could not open file for reading/writing.")

def reconstruct_insert(data):
    iv_buffer[::len(data),:] = data

    running = True
    batch_size = NON_CONTACT_NUM_LINES if (meter_type==TYPE_NON_CONTACT) else CONTACT_NUM_LINES
    raw_i, iv_i = 0, 0 # size OR next insertion index

    # reconstruct in batches
    while running:
        # read input file
        num_lines = 0
        for i in xrange(batch_size):
            line = fin.readline().strip()
            if line == "" or line is None: # (possibly) reach end of file
                running = False
                break
            num_lines += 1
            raw_buffer[raw_i+i,:] = line.split() # automatic casting
            raw_i += num_lines

        # reconstruct batch
        sensor_rows_processed = reconstructor.process(raw_buffer[:,raw_i], None, None, reconstruct_insert, False)
        raw_buffer = np.roll(raw_buffer, -sensor_rows_processed, axis=1)
        raw_i, iv_i = raw_i - sensor_rows_processed, sensor_rows_processed
        assert(raw_i >= 0 and iv_i >= 0)

        # write output file
        for i in xrange(iv_i):
            cols = ["%i" % iv_buffer[i,0]] + ['{:.6e}'.format(iv_buffer[i,c]) for c in xrange(iv, iv_buffer.shape[1]])
            fout.write(" ".join(cols) + os.linesep)
        iv_i = 0
fout.close()

# /*---------- PREP via NilmDB ----------*/

def prep_sensors(meter_name, path_file):
    # retrieve vref index
    try:
        (settings_meter, settings_calibration) = load_configs(meter_name)
        col_vref = settings_meter["sensors"]["voltage"]["sensor_index"]
        idx_sensors = settings_meter["sensors"]["current"]["sensor_indices"]
    except KeyError as e:
        fail("Error reading config for [%s], missing [%s]." % (meter_name,e[0]))

    # import raw sensor data
    nilmdb_create(nilmdb_path("sensors-raw"), "int32_8")
    nilmdb_insert(nilmdb_path("sensors-raw"), path_file)

    # run sinefit/prep on sensors
    nilmdb_create(nilmdb_path("sensors-sinefit"), "float32_3")
    nilmdb_sinefit(nilmdb_path("sensors-raw"), nilmdb_path("sensors-sinefit"),
                  col_vref+1)

    sensors_prep_names = list()
    for i, s_i in enumerate(idx_sensors):  # all sensors present in raw data
        col = 1 + s_i  # currents cols are even in NilmDB
        sensors_prep_names.append("sensors-prep-%i" % i)
    nilmdb_prep_dst = nilmdb_path(sensors_prep_names[-1])
    nilmdb_create(nilmdb_prep_dst, "float32_8")
    nilmdb_prep(nilmdb_path("sensors-raw"), nilmdb_prep_dst,
                nilmdb_path("sensors-sinefit"), col, 0)

    # extract PREP streams
    for name in sensors_prep_names:
        nilmdb_extract(nilmdb_path(name), tmpfile_path(name))

    # cleanup NilmDB streams
    nilmdb_destroy(nilmdb_path("sensors-raw"))
    nilmdb_destroy(nilmdb_path("sensors-sinefit"))
    for name in sensors_prep_names:
        nilmdb_destroy(nilmdb_path(name))

    return sensors_prep_names

def prep_currents(meter_name, file_iv, files_prep=None):
    (settings_meter, settings_calibration) = load_configs(meter_name)
    try:  # cols are 0-indexed here, but 1-indexed for NilmDB
        num_phases = settings_meter["phases"]
        sinefit_rotations = settings_calibration["sinefit_rotations"]
        if settings_meter["type"] == TYPE_NON_CONTACT:
            col_vref = 0
cols_curr = range(1, num_phases + 1)
cols_total = num_phases + 1
has_neutral = settings_meter['calibration']['has_neutral']
if not has_neutral:
sinefit_rotations = settings_calibration['full_sinefit_rotations']
elif settings_meter['type'] == TYPE_CONTACT:
    vref_letter = settings_meter['sensors']['voltage']['sinefit_phase']
col_vref = ord(vref_letter.lower()) - ord('a')  # A, B, C
assert(col_vref <= num_phases)
cols_curr = range(num_phases, 2*num_phases)
cols_total = 2*num_phases
except KeyError as e:
    fail("Error reading config for %s, missing %s." % (meter_name, e[0]))

if files_prep is not None and len(files_prep) != num_phases:
    fail("Number of PREP filenames must match the number of phases.")

# decide stream names
stream_iv = nilmdb_path("%s-iv" % meter_name)
stream_sinefit = nilmdb_path("%s-sinefit" % meter_name)
streams_prep = list()  # populated later

# import iv data into NilmDB
nilmdb_create(stream_iv, "float32_%i" % cols_total)
nilmdb_insert(stream_iv, file_iv)

# run sinefit/prep on sensors
nilmdb_create(stream_sinefit, "float32_3")
nilmdb_sinefit(stream_iv, stream_sinefit, col_vref+1)
for i, c in enumerate(cols_curr):
    streams_prep.append("%s-prep-%i" % (meter_name, i))
    stream_prep_dst = nilmdb_path(streams_prep[-1])
    nilmdb_create(stream_prep_dst, "float32_8")
    nilmdb_prep(stream_iv, stream_prep_dst, stream_sinefit, c+1,
    → sinefit_rotations[i])

# extract PREP streams
for i, name in enumerate(streams_prep):
    dst_path = tmpfile_path(name) if files_prep is None else files_prep[i]
    nilmdb_extract(nilmdb_path(name), dst_path)

# cleanup NilmDB streams
nilmdb_destroy(stream_iv)
nilmdb_destroy(stream_sinefit)
for name in streams_prep:
    nilmdb_destroy(nilmdb_path(name))

return streams_prep

# /*---------- Modify YML Files ----------*/
# override NC vref in meters.yml with strongest voltage sensor, determine voltage scaling

def autoselect_vref(meter_name, sensor_path, autoselect=True):
    # detect strongest voltage sensor
    execute("head -n 10000 %s > %s" % (sensor_path, tmpfile_path("sensors-raw-sample"))
    sensors_sample = load_data(tmpfile_path("sensors-raw-sample"))

    if autoselect:
        # find largest voltage
        i_vref, rms_vref = max_vrms(sensors_sample)

        # read in meters.yml file
        yml_str = None
        try:
            with open(METERS_FILE, 'r') as f:
                yml_str = f.read()  # avoid using PyYAML to preserve commenting
        except IOError:
            fail("Can't load meters file at [%s]." % METERS_FILE)

        # find/replace sensor index manually - preserve commenting
        yml_split = yml_str.split(os.linesep)
        for i in xrange(len(yml_split)):
            if yml_split[i] == '': continue
            comm_idx = yml_split[i].find('#')
            if comm_idx != -1:  # remove line/inline comments
                yml_split[i] = yml_split[i][:comm_idx].ljust(len(yml_split[i]))
        yml_stripped = os.linesep.join(yml_split)

        finder = re.finditer("meter\d+:", yml_stripped)
        meters_idx = [m.start(0) for m in finder] + [len(yml_stripped)]

        yml_revised = None
        for i in xrange(len(meters_idx) - 1):
            block_start, block_end = meters_idx[i], meters_idx[i + 1]
            meter_block = yml_stripped[block_start:block_end]
            finder = re.finditer("(%s)::sensors.voltage.sensor_index: (\d+)" % meter_name, meter_block, re.MULTILINE|re.DOTALL)
            intervals = [m for m in finder]
            if len(intervals) == 1 and meter_name == intervals[0].group(1):
                i_end = intervals[0].end(0)
                current_vref = intervals[0].group(2)
                i_before, i_after = (block_start + i_end - len(current_vref)), (block_start + i_end)
                yml_revised = yml_str[:i_before] + str(i_vref) + yml_str[i_after:]
            break

    if yml_revised is None:
        fail("Non-contact meter could not be located within meters.yml file.")
# write back to meters.yml file
try:
    with open(METERS_FILE, 'w') as f:
        f.write(yml_revised)
except IOError:
    fail("Could not write to meters.yml file.")
else:
    # keep specified voltage reference index
    i_vref = meter_attr(meter_name, "sensors.voltage.sensor_index")
    dc_offset = np.mean(sensors_sample[1][:, i_vref])
    rms_vref = np.sqrt(np.mean(np.square(sensors_sample[1][:, i_vref] - dc_offset)))

# re-load meters.yml file
(settings_meter, settings_calibration) = load_configs(meter_name)

try:
    nominal_rms_voltage = settings_meter["sensors["voltage["nominal_rms_voltage"]
except KeyError as e:
    fail("Error reading config for [%s], missing [%s]." % (meter_name, e[0]))

scale_vref = float(nominal_rms_voltage) / float(rms_vref)

return scale_vref

def generate_calibration_file(meter_name, scale_vref, transformation, rotations, neutral_phase=None, converted=False):
    # construct calibration attributes
    settings_calibration = dict()
    settings_calibration["voltage_scale"] = float(scale_vref)
    if not converted:
        settings_calibration["sensor_matrix"] = transformation.T.tolist()

        # adjust rotations for digital integration
        rotations += 4*np.pi  # shift to [0, 2pi] range from [-pi, pi] range
        if meter_attr(meter_name, "sensors.voltage.digitally_integrate"):
            rotations -= 0.5 * np.pi  # voltage pickup offset from line by 90 deg
        rotations %= 2*np.pi

        current_matrix = np.linalg.pinv(transformation.T)
        full_current_matrix, full_rotations =
            complete_transformation(current_matrix, rotations, neutral_phase)

        settings_calibration["sinefit_rotations"] = rotations.tolist()
        settings_calibration["full_sinefit_rotations"] = full_rotations.tolist()
        settings_calibration["current_matrix"] = current_matrix.tolist()
        settings_calibration["full_current_matrix"] =
            full_current_matrix.tolist()
    else:
        settings_calibration["sinefit_rotations"] = rotations.tolist()
settings_calibration["full_sinefit_rotations"] = rotations.tolist()
settings_calibration["current_matrix"] = transformation.tolist()
settings_calibration["full_current_matrix"] = transformation.tolist()

# write to callibration file
try:
    CALIBRATION_DIR = "/opt/configs/meters"
    with open(CALIBRATION_DIR+"/"+meter_name+".yml",'w') as f:
        f.write(yaml.safe_dump(settings_calibration, default_flow_style=False, canonical=False))
    except IOError:
        fail("Could not write to calibration file for [\%s]."%meter_name)

if not converted:
    return (full_current_matrix, full_rotations)
else:
    return (transformation, rotations)

# /*---------- User Interactive Code ----------*/

def interactive_capture(meter_name, file_raw):
    (settings_meter, settings_calibration) = load_configs(meter_name)
    capture = Capture(meter_name, settings_meter, file_raw)
    capture.start()
    while capture.num_samples() == 0: # handle prelim errors in capture thread
        if not capture.thread.isAlive():
            sys.exit(1) # wait until something captured
    user_prompt = True
    def update_progress(capture):
        os.system('clear') # clear terminal
        print "Capturing from %s..." % meter_name
        print 'Progress: %i (%i seconds)' % (capture.num_samples(), capture.num_seconds())
        print "Type STOP to finish, ABORT to cancel."
        sys.stdout.write(readline.get_line_buffer())
        sys.stdout.flush()
        time.sleep(0.5)

        progress_thread = Thread(target = lambda: update_progress(capture))
        progress_thread.start()
        while capture.thread.isAlive():
            user_input = raw_input()
            if user_input.lower() == "stop":
                user_prompt = False
                progress_thread.join()
            print "Stopping capture..."
status = 0
break
if user_input.lower() == "abort":
    user_prompt = False
    progress_thread.join()
    print "Abortig capture..."
    status = 1
    break
    capture.stop()
    print 'Captured: %i (%i seconds)' % (capture.num_samples(),
                                           capture.num_seconds())
    return status

if __name__ == "__main__":
    DIR_TMP = "/tmp/submeter"
    if os.path.exists(DIR_TMP): # remove residual files
        shutil.rmtree(DIR_TMP) # precautionary cleanup
        os.mkdir(DIR_TMP)

    file_raw = parse_path("~/raw.dat", False)
    file_iv = parse_path("~/iv.dat", False)

    meter_name = "meter1"
    (settings_meter, settings_calibration) = load_configs(meter_name)
    interactive_capture(meter_name, file_raw)
    reconstruct_iv(meter_name, file_raw, file_iv)
    shutil.rmtree(DIR_TMP) # cleanup
import sys, os
import subprocess
import re

from shared import *
from helpers import warn, fail

# /*---------- Execute External Commands ----------*/
def execute(cmd_str):
    printDebug(cmd_str)
    p = subprocess.Popen(cmd_str, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    stdout, stderr = p.communicate()
    if stderr != "":
        raise Exception("External command encountered error: %s%s %s.linesep, stderr))
    return stdout

# /*---------- NILM Command Endpoints ----------*/
def nilmdb_destroy_tmp():
    if execute("nilmtool list /tmp_submeter*") != "": # destroy if exists
        execute("nilmtool destroy -R /tmp_submeter*")

def nilmdb_destroy(path_nilmdb):
    if path_nilmdb.find("/tmp_submeter_") != 0:
        fail("Attempted to destroy external NilmDB stream.")
    if execute("nilmtool list %s %s %s_nilmdb) != "": # destroy if exists
        execute("nilmtool destroy -R %s %s_nilmdb")

def nilmdb_create(path_nilmdb, layout):
    execute("nilmtool create %s %s %s (path_nilmdb, layout)

def nilmdb_insert(path_nilmdb, path_file):
    t_start = int(execute("head -1 %s %s %s %s_file).split()[0])
    t_end = int(execute("tail -50 %s | awk 'NF' | tail -1" %s %s_file).split()[0]) + 1
    execute("nilmtool insert -s @%i -e @%i %s %s %s %s %s %s %s %s %s") % (t_start, t_end, path_nilmdb, path_file)
```python
def nilmdb_sinefit(path_nilmdb_src, path_nilmdb_dst, col_vref):
    execute("nilm-sinefit -n -c %i %s %s" % (col_vref, path_nilmdb_src, path_nilmdb_dst))

def nilmdb_prep(path_nilmdb_src, path_nilmdb_dst, path_nilmdb_sinefit, col_target, rad_rotation):
    execute("nilm-prep -c %i -R %f %s %s %s" % (col_target, rad_rotation, path_nilmdb_src, path_nilmdb_sinefit, path_nilmdb_dst))

def nilmdb_interval(path_nilmdb):
    ret_str = execute("nilmtool intervals -T %s" % path_nilmdb)
    i_str = [match.start() for match in re.finditer("\[ \d+ -> \d+ \]", ret_str)]
    if len(i_str) != 1:
        raise Exception("Expecting exactly 1 NilmDB interval.")
    ret_split = ret_str.strip().split()
    return (int(ret_split[1]), int(ret_split[3]) + 1)  # t_start, t_end

def nilmdb_extract(path_nilmdb, path_file):
    t_start, t_end = nilmdb_interval(path_nilmdb)
    execute("nilmtool extract -s @%i -e @%i %s > '%s'" % (t_start, t_end, path_nilmdb, path_file))
```
from helpers import *

import matplotlib
matplotlib.use('TKAgg')
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from matplotlib.backends.backend_pdf import PdfPages
import numpy as np

def maximizeWindow():
    mng = plt.get_current_fig_manager()
    mng.resize(*mng.window.maxsize())  # maximize window

def setupFitPlot(data):
    fig = plt.figure(1)
    maximizeWindow()
    plt.suptitle("Sensor Scaling Factors", size='large')
    fig.canvas.set_window_title("Sensor Scaling Factors")
    grid = gridspec.GridSpec(data.phases, data.sensors)

    labels_sensor = ["Sensor %i" % i for i in xrange(data.sensors)]
    if data.neutral_phase is None:
        labels_phase = ["Phase %s" % chr(65+phase) for phase in xrange(data.phases)]
    else:
        labels_phase = ["Phase %s" % chr(65+phase) for phase in xrange(data.phases + 1)]
        labels_phase.pop(data.neutral_phase)

    for phase in xrange(data.phases):
        for sensor in xrange(data.sensors):
            ax = plt.subplot(grid[phase, sensor])
            plt.axhline(y=0, color='k', ls='-', lw=0.5)  # x axis
            plt.axvline(x=0, color='k', ls='-', lw=0.5)  # y axis
            # labels
            if sensor == 0:
                ax.text(-0.3, 0.5, labels_phase[phase], size='medium',
                        rotation='vertical', va='center', transform=ax.transAxes)
            if phase == 0:
                ax.set_title(labels_sensor[sensor], size='medium')
if phase == 0 and sensor == 0:
    plt.ylabel("Sensor (Counts)"")
    plt.xlabel("Current (A)"")

return (plt, grid)
def plotConclusion(data, sensor_rotations, rotation_offsets):
    # retrieve and orient data
diff_mag_main, diff_mag_sensors = data.diff_currents.copy(),
                                      data.diff_sensors.copy()
    # apply current diff signs to indicate transient direction (ON/OFF) -
    # follow P1 changes (no smudging necessary)
    for ph_i in xrange(diff_mag_main.shape[1]):
        diff_mag_main[:,ph_i] = np.abs(diff_mag_main[:,ph_i]) *
                np.sign(data.diff_preps_currents[ph_i][:,0])
    for ph_i in xrange(diff_mag_sensors.shape[1]):
        diff_mag_sensors[:,ph_i] = np.abs(diff_mag_sensors[:,ph_i]) *
                np.sign(data.diff_preps_sensors[ph_i][:,0])

    num_phases = diff_mag_main.shape[1]
    labels_phase = ["Phase %s" % chr(65+phase) for phase in
                  xrange(num_phases)]
    labels_phase_letters = ["%s" % chr(65+phase) for phase in
                           xrange(num_phases)]

    fig = plt.figure(2)
    maximizeWindow()
    fig.canvas.set_window_title("Calibration Summary")
    plt.suptitle("Calibration Summary", size='large')
    grid = gridspec.GridSpec(3, num_phases+1)

def plot_time(grid_space, data_arr, title):
    plt.subplot(grid_space)
    plt.plot(data_arr, linewidth=4, alpha=0.5, linestyle="-")
    plt.title(title)
    plt.xlabel("Time (Samples)"")
    plt.ylabel("Current (A)"")
    plt.legend(labels_phase)

def plot_angles(grid_space, angle, title):
    ax = plt.subplot(grid_space, polar=True)
    ax.set_title(title, size='medium', y=1.1)
    ax.set_yticklabels([])
    ax.set_theta_zero_location('N')
    ax.set_xticklabels(["%i\N{DEGREE SIGN}"%(angle_adjust(45*i), u'\N{DEGREE SIGN}') for
                        i in xrange(8)])
    ax.text(0,0, "%i\N{DEGREE SIGN}" % (np.rad2deg(angle), u'\N{DEGREE SIGN}')
ax.plot(np.ones(2)*angle, np.array([0,1]), linewidth=3) # unit vector in
          specified direction

def plot_rotations(grid_space, rotations, title):
```
ax = plt.subplot(grid_space, polar=True)
ax.set_title(title, size='medium', y=1.1)
ax.set_yticklabels([])
for rot in rotations:
    # draw line from edge-edge of polar plot
    compl = np.deg2rad(angle_adjust(np.rad2deg(rot) + 180))
    ax.plot(np.array([rot, rot, compl, compl]),
            linewidth=3)
plt.legend(labels_phase_letters, loc='center right',
           bbox_to_anchor=(-0.2, 0.5), ncol=1)
plt.tight_layout()

plot_time(grid[0, :], diff_mag_main, "Known Current - Magnitude Diffs")
plot_time(grid[1, :], diff_mag_sensors, "Reconstructed Current - Magnitude Diffs")
plot_rotations(grid[2, 0], sensor_rotations, "Sensor Rotations")
for ph_i in xrange(num_phases):
    plot_angles(grid[2, ph_i+1], rotation_offsets[ph_i], "Rotation Offset \n%s" % labels_phase[ph_i])
plt.tight_layout()

def plotCurrents(data):
    fig = plt.figure(3)
    maximizeWindow()
    fig.canvas.set_window_title("Phase Currents")
    plt.suptitle("Phase Currents", size='large')
    grid = gridspec.GridSpec((2*data.phases), 1)
    labels_phase = ["Phase %s" % chr(65+phase) for phase in xrange(data.phases)]

    def subplot_current(id, ts, stream, title):
        x_ts = (np.arange(ts.shape[0]))[:, None] * np.ones((1, stream.shape[1]))
        ax = plt.subplot(grid[id, :])
        ax.set_title(title, x=-0.1, y=0.5, size='medium', rotation='vertical',
                     va='center')
        ax.get_xaxis().set_ticks([])
        plt.plot(x_ts, stream - np.mean(median_filter(stream[:, 60, :], 5),
                 axis=0), "-", alpha=0.5)
        if id == 0:
            plt.legend(["P1", "Q1"])
            plt.xlabel("Time")
            plt.ylabel("Current (A)")

    for i in xrange(data.phases): subplot_current(i, data.ts_currents,
            data.preps_currents[i][:, :2], "Known\n%s" % labels_phase[i])
    for i in xrange(data.phases): subplot_current((i + data.phases),
            data.ts_sensors, data.preps_sensors[i][:, :2], "Reconstructed\n%s" %
            labels_phase[i])
```
next_plot_id = 10

def plotGeneral(suptitle, streams, titles, legend=None):
    assert(len(streams) == len(titles))

global next_plot_id
plt.figure(next_plot_id)
plt.suptitle(suptitle, size='large')
maximizeWindow()
next_plot_id += 1
grid = gridspec.GridSpec(len(streams), 1)

axes = list()
for (i, data) in enumerate(streams):
    data = data.reshape(data.shape[0], -1)  # force 2D
    adj_ts = np.arange(data.shape[0])
    x_ts = adj_ts[:, None] * np.ones((1, data.shape[1]))
    ax = plt.subplot(grid[i, :])
    axes.append(ax)
    ax.set_title(titles[i], x=-0.1, y=0.5, size='medium', va='center')
    plt.plot(x_ts, data, '.', alpha=0.5)
    if legend is not None and len(legend) == data.shape[1]:
        plt.legend(legend)

return axes

# Usages example:
# plotDebug((ts, multi_stream), (ts2, stream_array[:, None]))
def plotDebug(*streams):
    global next_plot_id
    plt.figure(next_plot_id)
    maximizeWindow()
    next_plot_id += 1
    grid = gridspec.GridSpec(len(streams), 1)

    for (i, elem) in enumerate(streams):
        if type(elem) == tuple:
            (ts, data) = elem
        else:
            data = elem
            adj_ts = np.arange(data.shape[0])
            data = data.reshape(data.shape[0], -1)  # force 2D
            x_ts = adj_ts[:, None] * np.ones((1, data.shape[1]))
            plt.plot(x_ts, data, '.', alpha=0.5)
            plt.legend(range(data.shape[1]))

    def plotShow():
        plt.show()
```python
def plotDebugDiffs(data):
    plotDebug(*([data.preps_currents[i][;,2] for i in xrange(data.phases)] +
               [data.diff_preps_currents[i][;,2] for i in xrange(data.phases)]))
    plotDebug(*([data.preps_sensors[i][;,2] for i in xrange(data.sensors)] +
               [data.diff_preps_sensors[i][;,2] for i in xrange(data.sensors)]))

def savePlots(filepath):
    pp = PdfPages(filepath)
    figs = [plt.figure(n) for n in plt.get_fignums()]
    for fig in figs:
        fig.set_size_inches(17, 11) # ledger size, landscape orientation
        fig.savefig(pp, format='pdf')
    pp.close()
```

```python
from __future__ import print_function
from datetime import datetime

def time_str():
    global DATETIME
    try:
        DATETIME
    except NameError:
        DATETIME = datetime.now().strftime("%y%m%d_%H%M%S%f")[:-4]  # date_time = YYMMDD_hhmss
        return DATETIME

def setDebug(debug_flag):
    global DEBUG
    DEBUG = debug_flag

def isDebug():
    global DEBUG
    try:
        DEBUG
    except NameError:
        DEBUG = False  # False if undefined
    return DEBUG

def printDebug(*debug_str):
    if isDebug():
        print(*debug_str)

def setFlag(flag, value):
    global FLAGS
    try:
        FLAGS
    except NameError:
        FLAGS = dict()
    FLAGS[flag] = value

def hasFlag(flag):
    global FLAGS
    try:
        FLAGS
```
except NameError:
    return False

return (flag in FLAGS)

def getFlag(flag):
    global FLAGS
    return FLAGS[flag]
C.2 NILM Management Console

Source Code C.9

Filename: main.py

Collaborators: Jennifer F. Switzer

Description: Main entry point for management console. Can be invoked from terminal or within OS application launcher.

```python
#!/usr/bin/env python

from tkinter import *
import ttk

from logtab import LogsTab
from nilmtab import NilmTab
from drivetab import DriveTab
from statustab import StatusTab
import threading
from nilmthread import *
from helperfunctions import *
from nilmwidgets import *

class NilmFrame(Frame):
    """NilmFrame class for NILM Management Interface

    Initializes main loop and periodically updates overall system state.
    Holds all tabs, as well as buttons/labels to indicate and manage overall system state
    (i.e. toggle capture and database).
    ""
    Extends:
        Frame (Tkinter widget)
    Variables:
        root {Tk root} -- The interface’s Tk root.
    ""
    def __init__(self, parent):
        Frame.__init__(self, parent)
        self.parent = parent
        self.grid()
        #set the frame to fill the whole screen, and save those dimensions
        #for setting relative sizes of child widgets
        self.w = self.parent.winfo_screenwidth()
        self.h = self.parent.winfo_screenheight()
        self.parent.title("NILM Management Console")
        self.parent.geometry("%dx%d+%d+%d" % (self.w, self.h, 0, 0))
```
# the focus keeps track of which tab is currently selected, is
# updated by polling functions
self.focus = "status"
# the state keeps tracking of whether capture and database are
# running/not running/failed
self.state = 0
# add widgets and tabs
self.add_widgets()
self.init_tabs()
# initialize polling function and then mainloop
self.poll()
self.mainloop()

def add_widgets(self):
  # initialize the main Tkinter frame
  self.mainPanel = Frame(self)
  self.mainPanel.grid()

  # add labels, indicator canvases and buttons for checking and
  # managing the current state of nilm-capture and the database
  statusbar = Frame(self.mainPanel,borderwidth=2,width=self.w)
  statusbar.grid(column=0,row=0,sticky="we")
  self.stateLabel1 = nilmLabel(statusbar,"Database not running",
                               fontsize=14)
  self.stateLabel1.pack(side=LEFT)
  self.stateCanvas1 = Canvas(statusbar, width=32, height=22,bd=0)
  self.stateCanvas1.pack(side=LEFT)
  self.statusIndicator1 = self.stateCanvas1.create_oval(7, 2, 25,
                                                       20,fill="grey")
  self.stateLabel2 = nilmLabel(statusbar,"Capture not running",
                               fontsize=14)
  self.stateLabel2.pack(side=LEFT)
  self.stateCanvas2 = Canvas(statusbar, width=32, height=22,bd=0)
  self.stateCanvas2.pack(side=LEFT)
  self.statusIndicator2 = self.stateCanvas2.create_oval(7, 2, 25,
                                                       20,fill="grey")
  self.dbButton = nilmButton(statusbar,text="Start Database",
                             command=self.toggleDatabase)
  self.dbButton.pack(side=RIGHT,fill=BOTH,expand=1)
  self.captureButton = nilmButton(statusbar,text="Start Capture",
                                   command=self.toggleCapture)
  self.captureButton.pack(side=RIGHT,fill=BOTH,expand=1)

def init_tabs(self):
  # initialize the main notebook
  nb = ttk.Notebook(self.mainPanel)
  nb.enable_traversal()
  nb.grid(column=0,row=2)

  # add all of the tabs to the notebook
self.statustab = StatusTab(nb,"Status",self)
nb.add(self.statustab,text='Status',underline=0)
self.drivetab = DriveTab(nb,"Manage drive",self)
nb.add(self.drivetab, text='Manage drive', underline=0)
self.logstab = Logstab(nb,"Logs",self)
nb.add(self.logstab,text="Logs",underline=0)

#this polling function is called every half second
#it updates the overall system state and updates the focus based on
#which tab is selected
#invokes or kills update functions for all tabs accordingly

def poll(self):
    self.getstate()
    root = self.parent
    if str(self.statustab) in str(root.focus_get()) or
    str(root.focus_get()) == ":."
        self.focus = "status"
        self.statustab.tg.invoke()
        self.drivetab.tg.kill()
        self.logstab.tg.kill()
    elif str(self.drivetab) in str(root.focus_get()):
        self.focus = "drive"
        self.drivetab.tg.invoke()
        self.statustab.tg.kill()
        self.logstab.tg.kill()
    elif str(self.logstab) in str(root.focus_get()):
        self.focus = "logs"
        self.logstab.tg.invoke()
        self.statustab.tg.kill()
        self.drivetab.tg.kill()
    else: #if no tab is currently selected, update all tabs
        self.focus = None
        self.statustab.tg.invoke()
        self.logstab.tg.invoke()
        self.drivetab.tg.invoke()
    self.after(500,self.poll)

def toggleCapture(self): #called when start/stop capture is pressed
    if self.state == 0 or self.state == 4: #if the capture is currently
    running
        executecommand(["sudo","service","nilm-capture","stop"]
        self.switch_one()
    elif self.state == 1 or self.state == 3 or self.state == 5: #if the
    capture is currently not running or failed
        executecommand(["sudo","service","nilm-capture","start"]
        self.switch_zero()
    self.getstate()

def toggleDatabase(self): #called when start/stop database is pressed
    if self.state == 1 or self.state == 5: #if the database is currently
    running
executeCommand(['sudo', 'apache2ctl', 'stop'])
self.switch_two()

elif self.state == 2 or self.state == 3:
    # if the database is currently not running
    executeCommand(['sudo', 'apache2ctl', 'start'])
    self.switch_one()
    self.getState()

# called when the database and capture are both running, updates button labels/states accordingly

def switch_zero(self):
    self.dbButton['text'] = "Stop Database"
    self.dbButton['state'] = "disabled"

    self.captureButton['text'] = "Stop Capture"
    self.captureButton['state'] = "normal"

    self.stateLabel1['text'] = "Database running"
    self.stateLabel2['text'] = "Capture running"
    self.stateCanvas1.create_oval(7, 2, 25, 20, fill="dark green")
    self.stateCanvas2.create_oval(7, 2, 25, 20, fill="dark green")

# called when the database is running and capture is stopped or failed, updates button labels/states accordingly

def switch_one(self):
    self.dbButton['text'] = "Stop Database"
    self.dbButton['state'] = "normal"

    self.captureButton['text'] = "Start Capture"
    self.captureButton['state'] = "normal"

    self.stateLabel1['text'] = "Database running"
    self.stateLabel2['text'] = "Capture running"
    self.stateCanvas1.create_oval(7, 2, 25, 20, fill="dark green")
    self.stateCanvas2.create_oval(7, 2, 25, 20, fill="grey")

# called when the database is stopped and capture is stopped or failed, updates button labels/states accordingly

def switch_two(self):
    self.dbButton['text'] = "Start Database"
    self.dbButton['state'] = "normal"

    self.captureButton['text'] = "Start Capture"
    self.captureButton['state'] = "disabled"

    self.stateLabel1['text'] = "Database not running"
    self.stateLabel2['text'] = "Capture not running"
    self.stateCanvas1.create_oval(7, 2, 25, 20, fill="grey")
    self.stateCanvas2.create_oval(7, 2, 25, 20, fill="grey")
# called when the database is stopped and capture is running, updates button labels/states accordingly

def switch_four(self):
    self.dbButton["text"] = "Start Database"
    self.dbButton["state"] = "disabled"

    self.captureButton["text"] = "Stop Capture"
    self.captureButton["state"] = "normal"

    self.stateLabel1["text"] = "Database not running"
    self.stateLabel2["text"] = "Capture running"
    self.stateCanvas1.create_oval(7, 2, 25, 20, fill="grey")
    self.stateCanvas2.create_oval(7, 2, 25, 20, fill="dark green")

    # getstate is called once when the home tab is first selected
    # It gets the current state of the nilm system by calling getSystemState
    # And uses this information to update the initial state of the db and
    # Capture buttons

def getstate(self):
    # get the current system state
    self.state = getSystemState()
    # call update function accordingly
    if self.state == 0:
        self.switch_zero()
    elif self.state == 1:
        self.switch_one()
    elif self.state == 2:
        self.switch_two()
    elif self.state == 3:
        self.switch_two()
    elif self.state == 4:
        self.switch_four()
    else:
        self.switch_one()

    # update the state of all child tabs
    self.statustab.state = self.state
    self.drivetab.state = self.state
    self.logstab.state = self.state

def main():
    root = Tk()
    nb = NilmFrame(root)

    if __name__ == '__main__':
        main()
```python
#!/usr/bin/env python

from Tkinter import *
import ttk
from math import floor

from helperfunctions import *
from nilmtab import NilmTab
from nilmthread import ThreadGroup
from nilmwidgets import nilmButton
from meters import meter

class LogsTab(NilmTab):
    def __init__(self,parent,text,master):
        """Log tab class for NILM Management Interface.

        Allows user to view and export log files.

        Arguments:
        
        parent {Tkinter frame} -- The frame that holds the tab.
        text {String} -- Text to be displayed on the tab itself.
        master {Tkinter frame} -- The main frame of the application,
        used for getting focus
        """
        NilmTab.__init__(self,parent,text,master)
        self.selectedbutton = None #indicates which button was most recently
        """selected"
        self.master = master #save reference to the main frame, used to
        """check for focus"
        self.addWidgets()

    #executes once, when program is opened, adds all widgets to the log tab
    def addWidgets(self):
        #logframe that holds buttons to select which logs to print
        logframe = ttk.Labelframe(self,text="Log Files",width=int(floor(0.15*self.w)),height=int(floor(0.8*self.h)))
        logframe.grid(row=0,column=0,rowspan=2,sticky="ns")
        logframe.columnconfigure(0,weight=10)
        logframe.grid_propagate(False)
```

Filename: logtab.py

Collaborators: Jennifer F. Switzer

Description: Allows users to view NILM log files. Logs can be manually browsed or automatically updated.
#checkbutton that allows the user to toggle between a live and static feed
self.LiveOnOff = False
self.c = Checkbutton(logframe, text="Live",
variable=self.LiveOnOff, onvalue=True, offvalue=False, command=self.toggle)
self.c.grid(row=0)

#button that outputs the supervisor logs when pressed
supervisorButton = nilmButton(logframe, text='Supervisor', command=lambda: self.buttonselected("supervisor"))
supervisorButton.grid(row=1, sticky="we")

nilmdbButton = nilmButton(logframe, text="NilmDB intervals", command=lambda: self.buttonselected("db"))
nilmdbButton.grid(row=2, sticky="we")

#frame holds the text box
txtframe = Frame(self, width=int(floor(0.8*self.w)), height=int(floor(0.8*self.h)))
txtframe.grid(row=0, column=1)
txtframe.columnconfigure(0, weight=10)
txtframe.rowconfigure(0, weight=10)
txtframe.grid_propagate(False)

#text box where the logs are output to
self.txt = Text(txtframe, state='disabled')
self.txt.grid(sticky="wens")

#scroll bar to allow vertical scrolling through the textbox - only enabled when live is not selected
self.vscroll = Scrollbar(txtframe, orient=VERTICAL, command=self.txt.txt['yview'])
sel.txt['yscroll'] = self.vscroll.set
self.vscroll.grid(row=0, column=1, sticky="ns")

#button that zips up the log and config files when pressed, and allows to user to save them to a directory of their choice
self.extractbutton = nilmButton(self, text="Extract", command=self.extract)
self.extractbutton.grid(row=1, column=1, sticky="e")

#get all of the meters present in the logs file, and create buttons for them
meterNames = getMeters()
meters = {}
for m in meterNames.keys():
meterButton = nilmButton(logframe,
text=meterNames[m], command=meter(meterNames[m], self).printlog)
meterButton.grid(row=3+int(m), sticky="we")
meters[m] = meterButton
def printStaticLog(self):
    # enable the text box and delete its current contents
    self.txt["state"] = "normal"
    self.txt.delete("1.0",END)

    # determine which logs to print based on the value of selectedbutton
    logs = []
    if self.selectedbutton == 0: # supervisor log
        logs = getandreversefile("/var/log/nilm/supervisor.log")
    elif self.selectedbutton == -1: # nilmDB intervals
        logs = getNilmDBIntervals()
        self.txt.insert("1.0",logs)
        self.txt["state"] = "disabled"
        return
    elif self.selectedbutton != None: # one of the meters
        logs = getandreversefile("/var/log/nilm/meter" + str(self.selectedbutton) + ".log")

    # print the logs to the textbox and then disable it again
    for line in logs:
        self.txt.insert("1.0",line)
        self.txt["state"] = "disabled"

    # executes once every second when the live checkbutton is selected and focus is on logs tab
    # continually refreshes the content of the logs textbox
    def printLiveLog(self):
        if self.master.focus != None and self.master.focus != "logs": # if the focus is on another tab
            self.LiveOnOff = False
            self.c.deselect()
            self.vscroll.grid(row=0,column=1,sticky="ns")
            self.printStaticLog()
        elif self.LiveOnOff == True: # if the focus is on logs tab and the live checkbutton is selected
            # enable the text box and delete any text it currently contains
            self.txt["state"] = "normal"
            self.txt.delete("1.0",END)
            # check the value of selectedbutton to determine which logs file to output, and print them
            if self.selectedbutton == 0:
                (output,err) = executecommand(['"tail","-n","100","/var/log/nilm/supervisor.log"'])
                lines = reversed(output.strip().split(os.linesep))
                self.txt.insert("1.0",os.linesep.join(lines))
            elif self.selectedbutton == -1:
                output = getNilmDBIntervals()
                self.txt.insert("1.0",output)
                self.txt["state"] = "disabled"
                self.after(1000,self.printLiveLog)
return
else:
    (output, err) =
        executecommand(['tail', '-n', '100', '/var/log/nilm/meter' + 
                    str(self.selectedbutton) + '.log'])
    lines = reversed(output.strip().split(os.linesep))
    self.txt.insert('1.0', os.linesep.join(lines))
    #disable the text box, and set to execute again in 1 second
    self.txt['state'] = "disabled"
    self.after(1000, self.printLiveLog)

    #executes when extract button clicked, extracts log & config files to
    #location specified by user
    def extract(self):
        self.extractbutton['state'] = "disabled"
        extractdir()
        self.extractbutton['state'] = "normal"

    #executes when live checkbutton is clicked, toggles value of
    #self.liveOnOff
    def toggle(self):
        if self.LiveOnOff == False:
            #if live was just selected, remove the scroll bar and call
            printLiveLog
            self.LiveOnOff = True
            self.vscroll.grid_remove()
            self.printLiveLog()
        else:
            #if live was just deselected, restore the scroll bar and call
            printStaticLog
            self.LiveOnOff = False
            self.vscroll.grid(row=0, column=1, sticky="ns")
            self.printStaticLog()

        #executes when user selects log to print, updates value of
        #self.selectedbutton
        def buttonselected(self, name):
            #set the value of selectedbutton depending on which button is
            #selected
            if name == "supervisor":
                self.selectedbutton = 0
            elif name == "db":
                self.selectedbutton = -1
            else:  #meter selected
                self.selectedbutton = int(name[-1])
            #whether or not live selected, print logs statically
            #so that user sees the logs update immediately
            self.printStaticLog()
Source Code C.11

Filename: helperfunctions.py

Collaborators: Jennifer F. Switzer

Description: Stateless helper functions. Mostly used to interact with shell and filesystem to retrieve and set configurations.

```python
#!/usr/bin/env python

import os
import re
import yaml
import subprocess
import tkFileDialog
from distutils.dir_util import copy_tree, remove_tree
import shutil
import time
from nilmDialogBox import NilmDialog, MultipleOptionDialog, NopeYouCant
from prettytable import PrettyTable

def checkLogForErr(meter):
    """Checks if there are any error messages in the log for the specified meter""
    # read in the file
    info = executecommand(['tail', '-n', '4', '/var/log/nilm/' + meter + '.log'])[0]
    # check for error
    s = info.find("error")
    if s == -1:
        return None
    info = info[s:]
    e = info.find(r"\n")
    info = info[:e]
    return info

def getAllMeters():
    """Finds all connected meters and all meters configured in the meters.yml file, and generates error messages for them.""
    # read in meters.yml as a yaml object
    allMeters = {}
    meters_in_yml = {}
    with open('/home/nilm/Desktop/meters.yml', 'r') as yamlfile:
        text = yamlfile.read()
    yml_object = yaml.load(text)
    if yml_object is None: yml_object = {}
for meter in yml_object.keys():
    if yml_object[meter]["type"] == "noncontact":
        meters_in_yml[yml_object[meter]["serial_number"]] = meter
    else:
        ip_address = yml_object[meter]["ip_address"]
        ping_test = executecommand(["ping", "-w", "1", "ip_address","-c","1"])
        if len(re.findall(r"100\% packet loss",ping_test[0])) == 0:
            allMeters[meter] = "IP address"
        else:
            allMeters[meter] = "Error: IP address for this contact meter is not responding."

#get the meters that the nilm system has detected as connected
configured_meters = set(meters_in_yml.keys())
connected_meters = set()
if os.path.isdir("/dev/nilm"):
    devices = os.listdir("/dev/nilm")
    for d in devices:
        end = d.find("-")
        meter_name = d[:end]
        connected_meters.add(meter_name)

OKmeters = configured_meters.intersection(connected_meters) #meters that are both connected and in yml
config_not_connected = configured_meters.difference(OKmeters)
#meters that are configured but not connected
connected_not_configured = connected_meters.difference(OKmeters)
#meters that are connected but not configured

#iterate through connected + configured meters, checking if enabled + if error present in logs
for meter in OKmeters:
    log_error = checkLogForErr(meters_in_yml[meter])
    if not yml_object[meters_in_yml[meter]]["enabled"]: #if not enabled
        allMeters[meters_in_yml[meter]] = "Error: Meter is not enabled."
    elif log_error != None: #if an error has been detected in the logs file
        allMeters[meters_in_yml[meter]] = log_error
    else: #if no error has been detected
        allMeters[meters_in_yml[meter]] = ""

#iterate through configured + not connected meters, checking if enabled
for meter in config_not_connected:
if not yml_object[meters_in_yml[meter]]['enabled']:
    allMeters[meters_in_yml[meter]] = "Error: Meter is not enabled."
else:
    allMeters[meters_in_yml[meter]] = "Error: Meter is configured in the meters.yml file but is not connected."

# iterate through connected + not configured, setting error messages
# no need to check for enabled or logs errors, unconfigured meters aren't present in meters.yml or logs
for meter in connected_not_configured:
    allMeters[meter] = "Error: Meter is connected but is not configured in the meters.yml file."
return allMeters

def getNumCores():
    """Returns number of cores in device."""
    return int(executecommand(["nproc"])[0])

def ymlOK():
    """Checks for configuration errors in the meters.yml file."""
    if getSystemState() == 2 or getSystemState() == 3:
        return ""
    else:
        check_config = executecommand(["nilm-check-config"])""
        errorMsg = "[CONFIGURATION_ERROR]"
        if errorMsg in check_config:
            start = check_config.find(errorMsg) + len(errorMsg) + 2
            splice = check_config[start:]
            end = splice.find("\n")
            return splice[:end]
        return ""

def getCPUuse(numCores):
    """Returns the CPU use of each core in the device."""
    all_cores = {}
    # find percent idle for each core
    raw_text = executecommand(["mpstat", "-P", "ALL", "$1", "$1"])[0]
    idles = re.findall(r"\d*\.\d*\n", raw_text)[1:numCores+1]
    i = 1
    # reformat percent idle to percent use
    for string in idles:
        e = string.find("\n")
        use = 100 - float(string[:e])
        all_cores[i] = use
        i += 1
    return all_cores

def getDiskUse(mountpoint):
"""Returns the percent use of the disk mounted to mountpoint"""
info = executecommand(['df', mountpoint])[0]
try:
    percentUse = re.findall(r'\d\%', info)[0]
except IndexError:
    percentUse = "Unknown"
return percentUse

def backupMeters():
    """Creates a backup of the meters file if one does not already exist"""
    if not os.path.isfile("/opt/configs/metersBackup.yml") :
        temp_file = open("/opt/configs/metersBackup.yml", "w+")
        shutil.copyfile("/opt/configs/meters.yml",
                        "/opt/configs/metersBackup.yml")

def editMeters():
    """Opens the meters file in a text editor"""
    executecommand(['gedit', '/opt/configs/meters.yml'])

def configMeters(parent):
    """Lets the user edit the meters.yml file, applying changes only if no configuration errors arise."""
    if getTimeState() == 1 or getTimeState() == 5: #if db running, capture stopped
        backupMeters()
        #let the user configure the original meters file
        editMeters()
        #then check the new configuration
        if ymlOK() == "": #if there are no syntax errors in meters.yml
            #delete the backup file
            os.remove("/opt/configs/metersBackup.yml")
        else:
            d = NilmDialog(parent, "Error in meters configuration",
                            "Syntax error in meters.yml file. Would you like to retry or discard your changes?",
                            buttonLabels=('Retry', 'Discard changes'))
            if d.result == "Retry":
                #try again, maintaining the original,
                configMeters(parent)
            else:
                #restore the meters file to the backup
                shutil.copyfile("/opt/configs/metersBackup.yml",
                                "/opt/configs/meters.yml")
                #delete the backup
                os.remove("/opt/configs/metersBackup.yml")
        else:
            #...
d = NopeYouCan't(parent, "Database must be running and capture must be stopped to configure the meters file")

```python
def NilmScope(parent, meter):
    """Runs nilmscope on the specified meter for the streams chosen by the user."""
    if getSystemState() == 1 or getSystemState() == 5:  # if the db is running, and capture not running
        # check if meter is a contact or noncontact
        ContactMeterEh = False
        with open('/home/nilm/Desktop/meters.yml', 'r') as ymlfile:
            text = ymlfile.read()
            yml_object = yaml.load(text)

            if yml_object[meter.lower()]['type'] == "contact":  
                ContactMeterEh = True

            meterErrMsgs = getAllMeters()  
            if meterErrMsgs[meter] != "":  
                d15 = NopeYouCan't(parent, meterErrMsgs[meter] + "Nilm-scope cannot run properly on this meter until the problem is fixed.")
                return

        # let the user choose if they want to view current, voltage, or both
        d2 = MultipleOptionDialog(parent, ["Current", "Voltage", "Both"], "Select a trace")
        if ContactMeterEh:
            c = set([3, 4, 5])
            v = set([0, 1, 2])
        else:
            c = set([1, 3, 5, 7])
            v = set([0, 2, 4, 6])

            # generate the command string and run nilmscope
            if d2.result == "Current":  
                t = [str(i) for i in c]
            elif d2.result == "Voltage":  
                t = [str(i) for i in v]
            else:
                t = [str(i) for i in v.union(c)]

            t.sort()
            cmdstring = ["nilm-scope", meter, "-c"] + t
            executeCommand(cmdstring)
        else:
            d = NopeYouCan't(parent, "Database must be running and capture must be stopped to use NILM Scope")

    def dbRunning():
        """Checks if the database is running"""
```python
r = subprocess.Popen(['nilmtool', 'list', '/kitty/cat'], stderr=subprocess.PIPE)
err = r.communicate()[1]
if err == '':
    return True
else:
    return False

def calibrate(meter):
    """Runs nilm-calibrate on the specified meter in a new terminal""
    executecommand(['xterm', '-fa', 'Monospace', '-fs', '18', '-e', 'bash -c
        \"nilm-calibrate "+meter.lower()+"; exec bash\"'])

def getMeters():
    """Finds all meters present in the log files, to determine which
    logs to make available for viewing""
    logfiles = os.listdir('/var/log/nilm')
    meters = {}
    i = 2
    for f in logfiles:
        #create a button for each meter, and save a reference to it
        if f.startswith('meter') and f.endswith('.log'):
            meters[f[5]] = "Meter"+f[5]
            i+=1
    return meters

def getandreversefile(filepath):
    """Gets the file at filepath, and returns an array containing the
    file's lines in reverse order""
    f = open(filepath, 'r')
    arr = []
    while True:
        line = f.readline()
        if line == '':
            f.close()
            break
        arr.append(line)
    return arr

def getNilmDBIntervals():
    """Finds all streams in the nilm system and the intervals for which
    they are available
    Returns this info formatted into a table."
    intervals = ""
    r = executecommand(['nilmtool', 'list', '-En'])
    while True:
```

try:
    n = re.findall("/.*\n",r)[0][:-1]
except IndexError:
    break
s = len(n)+1
r = r[s:]
s = r.find("interval extents: ")
r = r[s:]
e = r.find("total")
i = r[:e-1]
intervals = intervals + "\n" + n + " " + i
r = r[e:]
l = intervals.splitlines()
#use the information found above to create a table
rows = [['Stream name','Interval extent']]  
for line in l:
    s = line.find("interval")
    if s != -1:
        e = s + 17
        name = line[:s]
        interval = line[e:]
        rows.append([name,interval])
    t = PrettyTable(rows)  #the PrettyTable class creates a formatted table
    return t.table

def extractdir():
    
    """Extracts the nilm log and config files to the location specified by the user."""
    #get the default file name
    default = "nilm-"+time.strftime("%y%m%d-%H%M%S")
    #let the user choose a destination folder through the tk file dialog
    destination = tkFileDialog.asksaveasfilename(initialdir="~",defaultextension=".zip",initialfile=default,title="Create a zip file to extract to")
    #get the selected destination folder
    if destination != ():
        destination = destination[0:-4]
        try:
            start = [m.start() for m in re.finditer('/',destination)][-1]
        except TypeError:
            return
        destFile = destination[start:]
        #make a temporary folder
        if not os.path.exists("temp"):
            os.makedirs("temp")
            os.makedirs("temp"+destFile)
            os.makedirs("temp"+destFile+"/logs")
            os.makedirs("temp"+destFile+"/config")
# copy the log and config files into the temp folder

```python
import copy_tree
import shutil

#copy_tree("/var/log/nilm","temp"+destFile+"/logs")
#copy_tree("/opt/config","temp"+destFile+"/config")

# move the contents of the temporary folder to the specified destination
shutil.make_archive(destination, "zip","temp")
```

# remove the temporary folder
remove_tree("temp")

```python
def appendToFile(fileName,text):
    """Appends the specified text to the file specified by fileName, used to append lines to the fstab file while mounting partitions"""

    # adjust permissions to allow writing to FSTAB
    executecommand(["sudo", "chmod", "666", "/etc/fstab"])
    # append the new text to the file's current contents and write them to the file
    with open(fileName,"a") as fp:
        fp.write(text+os.linesep)
    # restore permissions for FSTAB
    executecommand(["sudo", "chmod", "644", "/etc/fstab"])

def removeFromFstab(UUID):
    """Removes the line pertaining to the specified UUID from the fstab file, used while unmounting partitions."""

    # read in the current contents of the fstab file
    with open("/etc/fstab", 'r') as myfile:
        data = myfile.read()
    # find the line pertaining to the given UUID ID
    start = data.find("UUID="+UUID)
    if start < 0: return # not found
    end = start + data[start:].find(os.linesep) + 1 # after end of line
    if end < 0: end = len(data) # to end of file

    # splice out that line
    newdata = data[:start] + data[end:]
    # adjust permissions to allow writing to FSTAB
    executecommand(["sudo", "chmod", "666", "/etc/fstab"])
    # write the new data to the fstab
    with open("/etc/fstab","w") as fp:
        fp.write(newdata)
    # restore permissions for FSTAB
    executecommand(["sudo", "chmod", "644", "/etc/fstab"])

def executeCommand(commandstring,inpts=None):
    """Executes the command defined by commandstring and returns its result"""
```

```bash
    sudo chmod 666 /etc/fstab
```
result = subprocess.Popen(commandstring, stdout=subprocess.PIPE, stdin=subprocess.PIPE)
    if inpts is not None:
        if type(inpts) == str:
            return result.communicate(input=inpts)
        for inpt in inpts:
            result.stdin.write("%s\n" % inpt)
            # if result.poll() is not None:
            #   print result.stdout.read()
        return result.communicate()

def getSystemState():
    
    """Determines whether or not nilm-capture and the database are running, and whether or not the capture has failed."

    Returns an integer indicating:
    0 - both running
    1 - db running, capture not running
    2 - neither running
    3 - capture failed, db not running
    4 - capture is running and db is not, considered failed
    5 - capture failed, db running"

    #init variables to keep track of state
    capt_running = False
    db_running = False

    #get the current status of nilm-capture
    capt_check = str(executeCommand(['"sudo","service","nilm-capture","status"'])[0])
    pattern = re.compile("Active:")
    loc = pattern.search(capt_check).end()
    splice = capt_check[loc:loc+10]

    #check if the database is running
    db_running = dbRunning()

    #check if nilm-capture has failed, and if it is running
    if "failed" in splice:  #if nilm-capture is in 'failed' state
        if db_running:  #if the database is running
            return 5
        else:  #if the database is not running
            return 3
    elif "inactive" not in splice:  #if nilm-capture is running
        capt_running = True

    #determine overall system state
    if capt_running == True and db_running == True:
        return 0
    elif capt_running == False and db_running == True:
return 1
elif capt_running == False and db_running == False:
    return 2
else:
    return 4
Source Code C.12
Filename: drivetab.py
Collaborators: Jennifer F. Switzer
Description: Allows user to view all drives, with the exception of main hard drive.
Enables user to unmount, format, and mount the drives as primary or secondary
partitions.
1

#!/usr/bin/env python

2
3
4
5

from Tkinter import *
import ttk
from math import floor

6
7
8
9
10

from
from
from
from

partition import *
nilmwidgets import *
nilmtab import NilmTab
nilmTable import NilmTable

11
12
13
14

class DriveTab(NilmTab):
def __init__(self,parent,text,master):
"""Drive tab class for NILM Management Interface

15
16
˓→
17
˓→

Allows user to view all drives (with the exception of main
hard drive),
as well as unmount them, format them, and mount them as
primary or secondary.

18

Arguments:
parent {Tkinter Frame} -- The frame that holds the

19
20
˓→

DriveTab.

˓→

itself.

text {String} -- Text to be displayed on the tab

21

22
˓→
23
24
25
˓→
26
˓→
27

master {Tkinter Frame} -- The application’s main
frame, used for getting focus.
"""
NilmTab.__init__(self,parent,text,master)
self.systempartitions = systemPartitions() #get a dictionary
containing relevant info. on all available partitions
self.partitions = self.systempartitions.partitions #get a
list of all available partitions
self.addWidgets()

28
29
˓→
30

#executes once, when the program is first opened, and adds all
widgets to the drive tab
def addWidgets(self):

31

217


# create a warning label, to indicate whether or not database/capture are running

txtFrame0 = Frame(self, width=int(round(0.95*self.w)), height=int(round(0.1*self.w)))
txtFrame0.grid(row=0, column=0)
txtFrame0.rowconfigure(0, weight=10)
txtFrame0.columnconfigure(0, weight=10)
txtFrame0.grid_propagate(False)
txtFrame = Frame(txtFrame0)
txtFrame.grid(row=0, column=0)
self.warningLabel = nilmLabel(txtFrame, "Nilm capture and database both stopped. Storage may now be configured.", fontsize=14, fg="dark green")
self.warningLabel.grid(row=0, column=0)

# create buttons to start/stop database and capture, as well as labels indicating what is mounted to primary and secondary, as well as the disk use of the primary

smalltxtFrame = Frame(txtFrame, highlightthickness=1, highlightcolor="black", highlightbackground="black")
smalltxtFrame.grid(row=1, column=0)

primFrame = Frame(smalltxtFrame)
primFrame.grid(row=0, column=0)

secFrame = Frame(smalltxtFrame)
secFrame.grid(row=0, column=1)
diskuse = self.systempartitions.root.getAvailSpace()
self.primLabel = nilmLabel(primFrame, "Primary: Main hard drive", fontsize=14, fg="red3")
self.primLabel.grid(row=0, column=0, sticky="nw")
self.primdiskuseLabel = nilmLabel(primFrame, "Available Space: "+diskuse, fontsize=11)
self.primdiskuseLabel.grid(row=1, column=0, sticky="nw")
self.unmtPrim = nilmButton(primFrame, text="Unmount Primary", state="disabled", width=20, height=2, fontsize=14, command=lambda:self.systempartitions.unmountPrimary(self))
self.unmtPrim.grid(row=2, column=0, sticky="n")

self.secLabel = nilmLabel(secFrame, "Secondary: Main hard drive", fontsize=14)
self.secLabel.grid(row=0, column=1, sticky="nw")
self.secdiskuseLabel = nilmLabel(secFrame, "Available Space: "+diskuse, fontsize=11)
self.secdiskuseLabel.grid(row=1, column=1, sticky="nw")
self.unmtSec = nilmButton(secFrame, text="Unmount Secondary", state="disabled", width=20, height=2, fontsize=14, command=lambda:self.systempartitions.unmountSecondary(self))
self.unmtSec.grid(row=2, column=1, sticky="n")

# create a main frame to hold the table of available partitions

self.edFrame = ttk.Labelframe(self, text="Manage Partitions")
```python
self.driveframe = Canvas(self.edFrame, 
    height=int(round(0.85*self.h)), 
    width=int(round(0.95*self.w)))
self.driveframe.grid()
self.driveframe.rowconfigure(0, weight=10)
self.driveframe.columnconfigure(0, weight=10)
self.driveframe.grid_propagate(False)
self.edFrame.grid(row=1, column=0)

# initialize the table of partitions, adding a scrollbar if necessary
self.initTable()
if len(self.table.partitions) > 8:
    self.addScrollBar()

def addScrollBar(self):
    self.drivescroll = Scrollbar(self.edFrame, orient=VERTICAL, 
    command=self.driveframe.yview)
    self.driveframe.config(yscrollcommand=self.drivescroll.set)
    self.drivescroll.grid(row=0, column=2, sticky="ns")

# executes if there are more than 8 partitions, and adds a scrollbar to the partition table if so

# executes when the program is first opened, and again if partitions are added or removed
# creates a NilmTable and populates it with information and widgets
self.systempartitions = systemPartitions()
self.table = NilmTable(self.driveframe, self.systempartitions, self) 
self.table.pack(side="top", fill="x")
self.table.populate()
self.currentShape = self.table.shape
self.buttons = self.table.buttons

# executes once every 0.5 seconds when the tab is selected, and updates the contents of the 
# NilmTable, as well as all other buttons or labels, based on the state of the system (capture/db running or not)
def update(self):
    self.systempartitions = systemPartitions() 
    self.table.setPartInfo(self.systempartitions)
    newShape = self.table.shape # check the new "shape" of the data, to see if it has changed
    if newShape != self.currentShape:
        self.currentShape = newShape
        self.table.destroy()
```

219
self.initTable()
else:  # otherwise, simply update the information in the table
table.updateTable(self.systempartitions)

# update the primary and secondary disk use labels,  
# and the labels indicating what is mounted to
~ primary/secondary
primdiskuse = self.systempartitions.root.getAvailSpace()
secdiskuse = self.systempartitions.root.getAvailSpace()
~ primary or secondary, data is just being stored on main hard drive
primlabel = "Main hard drive"  
seclabel = "Main hard drive"
if self.systempartitions.hasPrimary():  # if a disk is mounted
    self.primLabel["fg"] = "black"
    self.unmtPrim["state"] = "normal"
    primlabel = self.systempartitions.primary.name
    primdiskuse =
~ self.systempartitions.primary.getAvailSpace()
else:  # if no disk is mounted as primary
    self.primLabel["fg"] = "red3"
    self.unmtPrim["state"] = "disabled"
    if self.systempartitions.hasSecondary():  # if a disk is
        mounted as secondary
            self.unmtSec["state"] = "normal"
            seclabel = self.systempartitions.secondary.name
            secdiskuse =
~ self.systempartitions.secondary.getAvailSpace()
else:  # if no disk is mounted as secondary
    self.unmtSec["state"] = "disabled"
    self.primdiskuseLabel["text"] = "Primary: " + primlabel
    self.secdiskuseLabel["text"] = "Available Space: " +
~ primdiskuse
    self.secLabel["text"] = "Secondary: " + seclabel
    self.secdiskuseLabel["text"] = "Available Space: " +
~ secdiskuse

# check current state (an indication of whether or not the
database and capture are running)  
# and update whether or not storage can be configured
~ accordingly
if self.state == 2 or self.state == 3:  # if nilm-capture and
    the database are both stopped
    self.warningLabel["text"] = "NILM capture and
database both stopped. Storage may now be configured."
    self.warningLabel["fg"] = "dark green"
else:  # if either nilm-capture or database are running
    self.warningLabel["text"] = "Storage may not be
configured until both NILM capture and the database are stopped."
    self.warningLabel["fg"] = "red3"
```python
#!/usr/bin/env python

from helperfunctions import *

class meter():
    def __init__(self, name, master=None):
        """Meter class for NILM Management Interface
        Allows for management (calibration, etc) of the nilm meter specified by name.
        
        Arguments:
        name {String} -- The meter’s name (Meter1, Meter2, etc).
        master {Tkinter widget} -- The Tkinter widget to interact with for printlog, nilmscope.
        ""
        self.name = name
        self.master = master

    def calibrate(self):
        calibrate(self.name)

    def printlog(self):
        if self.master is not None:
            self.master.buttonselected(self.name)

    def nilmscope(self):
        NilmScope(self.master, self.name)
```

#!/usr/bin/env python

#Based on: https://stackoverflow.com/questions/11047803

import Tkinter as tk
from nilmwidgets import *

class SimpleTable(tk.Frame):
    def __init__(self, parent, r, c, shape):
        
        SimpleTable creates a basic table will c columns and r rows, with a
        number of subrows set by shape.
        For instance, if shape = [1,3], the first row will have one subrow,
        and the second will have three.
        
        Extends:
        Frame -- Tkinter frame widget.

        Arguments:
        parent {Tkinter frame} -- containing frame of the table
        r {int} -- number of rows
        c {int} -- number of columns
        shape {[int]} -- a list number of subrows indexed by row

        
        # use colored background to imitate grid lines
        tk.Frame.__init__(self, parent, background="dark green")
        self._widgets = []
        rowcount = 0
        for row in range(r):
            rowspan = shape[row]
            current_row = []
            #for the first column, create only one row
            label = tk.Frame(self,
                              borderwidth=0, width=10)
            label.grid(row=rowcount, rowspan=rowspan, column=0,
                        sticky="nsew", padx=1, pady=1)
            current_row.append(label)
            #for the rest of the columns, the number of rows is set by the
            #value of shape
            for column in range(1, c):
                sub_columns = []
                for i in range(rowspan):
label = tk.Frame(self,
    borderwidth=0, width=10)
label.grid(row=rowcount+i, column=column, sticky="nsew",
    padx=1, pady=1)
sub_columns.append(label)
if row == 0:
current_row.append(label)
else:
current_row.append(sub_columns)
rowcount = rowcount + rowspan
self._widgets.append(current_row)
for column in range(c):
    self.grid_columnconfigure(column, weight=1)

def set(self, row, column, subcolumn, value): #used for setting the
    contents of text-only cells
    parent = self.getparent(row,column,subcolumn)
    if isinstance(value, basestring):
        label = nilmLabel(parent,value,fontsize=14)
        label.grid()
        if len(value) < 10:
            label["width"] = 10
        return label

def getparent(self,row,column,subcolumn): #returns a pointer to the
    widget at the indicated loc
    try:
        if subcolumn == None:
            return self._widgets[row][column]
        else:
            return self._widgets[row][column][subcolumn]
    except IndexError:
        return None

class NilmTable(SimpleTable):
    def __init__(self,parent,systempartitions,master):
        """NilmTable initializes a SimpleTable with the shape determined by
        the structure of systempartitions.
        The populate function fills the table with the information given by
        systempartitions."
        self.master = master
        self.setPartinfo(systempartitions)
#initialize the table with given information
SimpleTable.__init__(self,parent,len(self.partitions)+1,7,self.shape)

#save a dictionary of partition:[widget] values
#to maintain pointers to all non-static widgets
self.widgets = {}

#also save a list of all the buttons in the table
#so that they can all be disabled if needed
self.buttons = []

#setPartInfo resets the information used to fill the table, and resets
#the table shape
#but doesn’t actually update the information displayed in the table
def setPartInfo(self,systempartitions):
    self.systempartitions = systempartitions
    self.partitions = systempartitions.drives
    #get the shape of the table from partition_info
    root = ''.join([i for i in self.systempartitions.root.name if not
                    i.isdigit()])[5:]
    if root in self.partitions:
        self.partitions.pop(root)
    self.drives = self.partitions.keys()
    self.drives.sort()
    self.shape = [1]
    for drive in self.drives:
        self.shape.append(len(self.partitions[drive]))

#updates all contents of the table
def updateTable(self,systempartitions):
    self.setPartInfo(systempartitions)
    for drive in self.partitions:
        for partition in self.partitions[drive]:
            self.updatepart(partition)

#populates the table with widgets and information
#called once, when table is first initialized
def populate(self):
    #first, fill in the header names of all columns
    self.set(0,0,None,"Drive")
    self.set(0,1,None,"Partition")
    self.set(0,2,None,"Mounted On")
    self.set(0,3,None,"Avail. Space")
    self.set(0,4,None,"Total Space")
    self.set(0,5,None,"Status")
    self.set(0,6,None,"Manage")

    #then, populate the rest of the information
    row = 1
    for drive in self.drives:
self.set(row, 0, None, drive)
subcolumn = 0
partitions = self.partitions[drive]
partitions.sort()
for partition in partitions:
    # save a pointer to all widgets
    self.widgets[partition.name] = []

    # set partition name column
    namelabel = self.set(row, 1, subcolumn, partition.name[5:])

    # create widgets for mount location frame
    parent = self.getparent(row, 2, subcolumn)
    mFrame = tk.Frame(parent)
    mFrame.grid()
    unmntButton = nilmButton(mFrame, fontsize=14, text="Unmount")
    unmntButton.grid(column=1, row=0)
    mnlable = nilmLabel(mFrame, ",", width=13, fontsize=14)
    mnlable.grid(column=0, row=0)

    # save pointers to the widgets
    self.buttons.append(unmntButton)
    self.widgets[partition.name].append((mnlable, unmntButton))

    # initialize a label for percent use, and save a pointer to it
    percentUseLabel = self.set(row, 3, subcolumn, ",")
    self.widgets[partition.name].append(percentUseLabel)

    # initialize a label for percent use, and save a pointer to it
    tsLabel = self.set(row, 4, subcolumn, ",")
    self.widgets[partition.name].append(tsLabel)

    # initialize a label for the status of the partition, and save a pointer to it
    statusLabel = self.set(row, 5, subcolumn, ",")
    self.widgets[partition.name].append(statusLabel)

    # create button widgets for managing the partition
    parent = self.getparent(row, 6, subcolumn)
    acbuttonFrame = tk.Frame(parent)
    acbuttonFrame.grid()
    mpButton = nilmButton(acbuttonFrame, fontsize=12, text="Use as Primary")
    mpButton.grid(row=0, column=0)
    msButton = nilmButton(acbuttonFrame, fontsize=12, text="Use as Secondary")
    msButton.grid(row=0, column=1)
    fmButton = nilmButton(acbuttonFrame, fontsize=12, text="Format")
fmButton.grid(row=0, column=2)

# Save pointers to all of the buttons
self.buttons.append(mpButton)
self.buttons.append(msButton)
self.buttons.append(fmButton)

self.widgets[partition.name].append((mpButton, msButton, fmButton))

subcolumn += 1

# Then configure all widgets based on the status of the partition
self.updatepart(partition)
row += 1

# Updatepart reconfigures the widget configs for the designated partition
# Should be called on all partitions after partition_info is reset
# In order for changes to actually be displayed

def updatepart(self, partition):
    drive = partition.name[:-1]

    # Save pointers to all widgets for this partition
    mountLabel = self.widgets[partition.name][0][0]
    mountButton = self.widgets[partition.name][0][1]
    usePercent = self.widgets[partition.name][1]
    totalspace = self.widgets[partition.name][2]
    status = self.widgets[partition.name][3]
    mpButton = self.widgets[partition.name][4][0]
    msButton = self.widgets[partition.name][4][1]
    fmButton = self.widgets[partition.name][4][2]
    nameLabel = self.widgets[partition.name][5]

    mountLabel["text"] = partition.getMountPoint()
    usePercent["text"] = partition.getAvailSpace()
    totalspace["text"] = partition.getTotalSpace()
    status["text"] = partition.state
    status["width"] = 30

    mountButton["command"] = lambda: partition.unmount(self.master)
    mpButton["command"] = lambda: partition.useAsPrimary(self.master)
    msButton["command"] = lambda: partition.useAsSecondary(self.master)
    fmButton["command"] = lambda: partition.formatext4(self.master)

    # Enable or disable buttons
if partition.canBeMounted:
    mpButton["state"] = "normal"
    msButton["state"] = "normal"
else:
    mpButton["state"] = "disabled"
msButton["state"] = "disabled"

if partition.canBeUnMounted:
    mountButton["state"] = "normal"
else:
    mountButton["state"] = "disabled"

if partition.canBeFormatted:
    fmButton["state"] = "normal"
else:
    fmButton["state"] = "disabled"

if self.systempartitions.hasPrimary() == True:
    mpButton["state"] = "disabled"

if self.systempartitions.hasSecondary() == True:
    msButton["state"] = "disabled"
Filename: nilmTab.py
Collaborators: Jennifer F. Switzer
Description: Parent class for all tabs.
from tkSimpleDialog import Dialog
from Tkinter import *

class NilmDialog(Dialog):
    def __init__(self, parent, title = None, text = None, buttonLabels = ("OK","Cancel")):
        """NilmDialog class for NILM Management Interface
        Basic dialog box class modified to allow for custom button labels, with custom return values.
        """
        self.text = text
        self.buttonLabels = buttonLabels
        Dialog.__init__(self, parent, title)

        #creates the button frame and populate it
        def buttonbox(self):
            box = Frame(self)
            self.b0 = Button(box, text=self.buttonLabels[0], width=10,
                            command=self.ok, default=ACTIVE)
            self.b0.pack(side=LEFT, padx=5, pady=5)
            self.b1 = Button(box, text=self.buttonLabels[1], width=10,
                            command=self.cancel)
            self.b1.pack(side=LEFT, padx=5, pady=5)

            #bind return/escape actions to OK/cancel functions
            self.bind("<Return>", self.ok)
            self.bind("<Escape>", self.cancel)
            box.pack()

        #creates the label widget to display the specified text
def body(self, master):
    dialog = Label(master, text=self.text)
    dialog.grid()

    # called when the 'OK' button - the button to the left - is selected
    def ok(self, event=None):
        self.result = self.buttonLabels[0]  # save the appropriate string as result
        if not self.validate():
            self.initial_focus.focus_set()  # put focus back
            return
        self.withdraw()
        self.update_idletasks()
        self.close()

    # called when the 'cancel' button - the button to the right - is selected
    def cancel(self, event=None):
        self.result = self.buttonLabels[1]  # save the appropriate string as result
        self.close()

    # returns focus to parent and destroys itself
    def close(self):
        self.parent.focus_set()
        self.destroy()

class MultipleOptionDialog(Dialog):
    def __init__(self, parent, buttonLabels, title=None, text=None):
        """Basic dialog box modified to allow for any number of options, each with their own unique return value"""
        self.text = text
        self.buttonLabels = buttonLabels
        self.result = None
        Dialog.__init__(self, parent, title)

        # creates the button frame and populate it
        def buttonbox(self):
            box = Frame(self)
            for button in self.buttonLabels:
                b = DialogButton(box, self, button, 20)
                b.pack(side=TOP, padx=5, pady=5)

            self.bind("<Escape>", self.close)
            box.pack()

        # creates the label widget to display the specified text
        def body(self, master):
            dialog = Label(master, text=self.text)
            dialog.grid()

            # returns focus to parent and destroys itself
def close(self):
    self.parent.focus_set()
    self.destroy()

class DialogButton(Button):
    def __init__(self, parent, box, text, width):
        """Special class of buttons for the MultipleOptionDialog, each with
        a command function that sets the result of,
        their parent dialog box to the appropriate value, and then returns
        focus to the main window & destroys the parent dialog box""
        super().__init__(self.parent, text=text, width=width,
                         command=self.cmd, height=3, font=('', 16))
        self.text = text
        self.box = box
    def cmd(self):
        self.box.result = self.text
        self.box.parent.focus_set()
        self.box.destroy()

class FormatDialog(NilmDialog):
    def __init__(self, parent, partition):
        """Dialog box specific to the drive tab, instantiated when the
        attempts to format a partition""
        message = "You are about to change the file system type of
        " + partition.name + " from " + partition.getFileSystem() + "
        to ext4. Continue?"
        super().__init__(self.parent, "Warning", message, ("OK", "Cancel"))

class UnmountDialog(NilmDialog):
    def __init__(self, parent, partition):
        """Dialog box specific to the drive tab, instantiated when the
        attempts to umount a partition""
        message = "You are about to umount " + partition.name + " from
        " + partition.getMountPoint() + ". Continue?"
        super().__init__(self.parent, "Warning", message, ("OK", "Cancel"))

class NopeYouCant(NilmDialog):
    def __init__(self, parent, text=None):
        """General purpose warning dialog that shows the user a warning
        message and gives only the option of pressing OK""
        if text is None:
            message = "Storage cannot be configured before the database and
            capture are stopped."
        else:
            message = text
        super().__init__(self.parent, "Warning", message)

def buttonbox(self):
    box = Frame(self)
self.b0 = Button(box, text=self.buttonLabels[0], width=10, 
command=self.ok, default=ACTIVE)
self.b0.pack(side=LEFT, padx=5, pady=5)
self.bind("<Return>", self.ok)
box.pack()
#!/usr/bin/env python

from Tkinter import *

class nilmButton(Button):
    def __init__(self, parent, text=None, command=None, fontsize=None, height=1, width=None, state="normal"):  
        """A regular Tkinter button, but with default configuration for stylistic purposes."""
        font = ("Helvetica", 14)
        if fontsize is not None:
            font = ("Helvetica", fontsize)
        Button.__init__(self, parent, text=text, command=command, bg="white", activebackground="green", font=font, height=height, width=width, state=state)

class nilmLabel(Label):
    def __init__(self, parent, text, fontsize=None, height=None, width=None, fg=None):
        """A regular Tkinter label, but with default configuration for stylistic purposes."""
        font = ("Helvetica", 14)
        if fontsize is not None:
            font = ("Helvetica", fontsize)
        Label.__init__(self, parent, text=text, font=font, height=height, width=width, fg=fg)
Source Code C.18

Filename: nilmthread.py

Collaborators: Jennifer F. Switzer

Description: Threading manager to run a process in the background.

```python
#!/usr/bin/env python

import threading
import time

# Based on: https://stackoverflow.com/questions/323972

class StoppableThread(threading.Thread):
    def __init__(self, target, sleeptime=1):
        r"""Thread class with a stop() method. The thread itself has to check regularly for the stopped() condition.

        Arguments:
            target {Function} -- The function to be called while the thread is running.
            sleeptime {int} -- The amount of time (in seconds) to sleep between function calls
        r"""

        super(StoppableThread, self).__init__()
        self._stop_event = threading.Event()
        self.target = target
        self.sleeptime = sleeptime

    def stop(self):
        self._stop_event.set()

    def stopped(self):
        return self._stop_event.is_set()

    def run(self):
        while not self.stopped():
            self.target()
            time.sleep(self.sleeptime)

class ThreadGroup():
    def __init__(self, targets, sleeptime=1):
        r"""A group of thread objects that are started and stopped as one."""
        self.targets = targets
        self.sleeptime = sleeptime
        self.threads = []
```
self.alive = False

def invoke(self):
    if not self.alive:
        for target in self.targets:
            thread = StopableThread(target, self.sleeptime)
            thread.start()
            self.threads.append(thread)
            self.alive = True

def kill(self):
    if self.alive:
        for thread in self.threads:
            thread.stop()
        self.threads = []
        self.alive = False
#!/usr/bin/env python
from helperfunctions import *
import re
from nilmDialogBox import *

class Partition():
    def __init__(self, name):
        # An object that allows for management (i.e. mounting, formatting) of the partition specified by name.
        self.name = name  # full file path, ex: /dev/sda1
        self.refreshState()

    # returns a string indicating the partition's mountpoint
    def getMountPoint(self):
        info = executecommand(['lsblk', self.name])[0]
        s = info.find(self.name) + len(self.name)
        info = info[s:]
        # check if a mountpoint was actually found
        try:
            mount_point = re.findall(r'/.*', info)[0]
            # if the name of the mountpoint is too long, shorten it
            if len(mount_point) > 13 and mount_point != '/opt/secondary':
                mount_point = mount_point[:10] + '...
        except IndexError:
            mount_point = "Not mounted"
        return mount_point

    # returns a string indicating the partition's available storage space
    def getAvailSpace(self):
        if self.getMountPoint() == "Not mounted":
            return "-
        else:
            # get the available space in kblocks of memory
            info = executecommand(['df', '--output=avail', self.name])[0]
            digits = re.findall(r'\d', info)
            try:
kblocksleft = float('').join(str(d) for d in digits)
except ValueError:
    return "-"
# convert to an easier to read format
if kblocksleft > 10**9:
gblocksleft = round(kblocksleft/1000000,2)
return str(gblocksleft)+'G'
else:
    mblocksleft = round(kblocksleft/1000,2)
return str(mblocksleft)+'M'

# returns a string indicating the partition's total storage space
def getTotalSpace(self):
    info = executecommand(['lsblk',self.name,'-o','SIZE'])[0]
    try:
        totalsize = re.findall(r'.\d\.\d.',info)[0]
    except IndexError:
        totalsize = ""
    return totalsize

# returns a string indicating the partition's UUID
def getUUID(self):
    info = executecommand(['lsblk','-f',self.name])[0]
    try:
        UUID = re.findall(r'\w{8}-\w{4}-\w{4}-\w{4}-\w{12}',info)[0]
    except IndexError:
        UUID = "Unknown"
    return UUID

# returns a string indicating the partition's file system
def getFileSystem(self):
    info = executecommand(['lsblk','-f',self.name])[0]
    name = self.name[5:]
s = info.find(name)+len(name)
info = info[s:]
file_system = re.findall(r'\S*',info)
fs = ''
i = 0
while fs == '' and i < 5:
    fs = file_system[i]
i += 1
if fs.isdigit() or fs == "":
    fs = "no file system detected"
return fs

# unmounts the partition from its current mountpoint
def unmount(self,parent): # parent is needed for dialog box
    if getSystemState() != 2 and getSystemState() != 3: # if database and/or capture are running
NopYouCant(parent)  #prevent the user from configuring storage
else:  #if database and capture are stopped
    original_mp = self.getMountPoint()
    d =UnmountDialog(parent,self)
    if d.result == "OK" and original_mp != "/":  #make sure partition is not mounted to root
        executecommand(['sudo', "umount", self.name])
        self.refreshState()
        #if partition was originally mounted as primary, remove its UUID from fstab
        if original_mp == "/opt/data":
            UUID = self.getUUID()
            removefromfstab(UUID)

#changes the partitions file system to ext4
    def formatext4(self,parent):  #parent is needed for dialog box
        if getSystemState() != 2 and getSystemState() != 3:  #if database and/or capture are running
            NopYouCant(parent)  #prevent the user from configuring storage
        else:  #if database and capture are stopped
            d = FormatDialog(parent,self)
            if d.result == "OK" and self.getMountPoint() != "/":  #make sure partition is not mounted to root
                fdisk_interaction = ['d']*20 + ['n', 'p', "1"] + ['"]*2 + ["w"]
                disk_path = self.name.strip("0123456789")
                part_path = disk_path + "1"
                executecommand(['sudo', "fdisk", disk_path], fdisk_interaction)
                executecommand(['sudo', "mkfs.ext4", "-q", part_path], "y")
                self.refreshState()

        #mounts the partition on /opt/data so that it can be used as primary storage
        def useAsPrimary(self,parent):
            if getSystemState() != 2 and getSystemState() != 3:  #if database and/or capture are running
                NopYouCant(parent)  #prevent the user from configuring storage
            elif self.getMountPoint() != "/":  #make sure partition is not mounted to root
                #mount at /opt/data and change ownership of the file
                executecommand(['sudo',"mount",self.name,"/opt/data"])
                executecommand(['sudo',"chown","-R","nilm:nilm","/opt/data"])
                #get the UUID
                drive = self.name[::]
UUID = self.getUUID()

#edit the fstab accordingly
appendToFile("/etc/fstab","UUID="+UUID+" /opt/data
\indent ext4 errors=remount-ro,data=journal 0 1")
self.refreshState()

#mounts the partition on /opt/secondary so that it can be used as
\indent secondary storage

def useAsSecondary(self,parent):
\indent if getSystemState() != 2 and getSystemState() != 3: #if
\indent \indent database and/or capture are running
\indent \indent NopeYouCant(parent) #prevent the user from
\indent \indent configuring storage
\indent \indent elif self.getMountPoint() != "/": #make sure partition is
\indent \indent \indent not mounted to root
\indent \indent \indent \indent #mount at /opt/secondary and change ownership of the
\indent \indent \indent \indent \indent file
\indent \indent \indent \indent \indent \indent executecommand(["sudo","mount",self.name,"/opt/secondary"])\indent \indent \indent \indent \indent \indent executecommand(["sudo","chown","-R","nilm:nilm",
\indent \indent \indent \indent \indent \indent \indent \indent \indent "/opt/secondary")
\indent \indent \indent self.refreshState()

#updates the state of the partition based on where its mounted and
\indent its file system

def refreshState(self):
\indent #self.state is a human readable description of the state of
\indent \indent \indent the partition
\indent \indent \indent self.canBeMounted, self.canBeUnmounted, and
\indent \indent \indent \indent self.canBeFormatted are boolean values describing possible actions
\indent mp = self.getMountPoint()
\indent fs = self.getFileSystem()
\indent \indent if mp == "/":
\indent \indent \indent self.state = "Mounted to root. Cannot be unmounted."
\indent \indent \indent self.canBeMounted = False
\indent \indent \indent self.canBeUnMounted = False
\indent \indent \indent self.canBeFormatted = False
\indent \indent elif mp == "/opt/data":
\indent \indent \indent self.state = "Mounted as primary."
\indent \indent \indent self.canBeMounted = False
\indent \indent \indent self.canBeUnMounted = True
\indent \indent \indent self.canBeFormatted = False
\indent \indent elif mp == "/opt/secondary":
\indent \indent \indent self.state = " Mounted as secondary."
\indent \indent \indent self.canBeMounted = False
\indent \indent \indent self.canBeUnMounted = True
\indent \indent \indent self.canBeFormatted = False
\indent \indent elif fs == "ext4" and mp == "Not mounted":
\indent \indent \indent self.state = "Unmounted, formatted correctly."
\indent \indent \indent self.canBeMounted = True
self.canBeUnMounted = False
self.canBeFormatted = False

eif fs != "ext4" and mp == "Not mounted":
    self.state = "Not formatted correctly."
    self.canBeMounted = False
    self.canBeUnMounted = False
    self.canBeFormatted = True
else:
    self.state = "Currently mounted."
    self.canBeMounted = False
    self.canBeUnMounted = True
    self.canBeFormatted = False

class systemPartitions():
    def __init__(self):
        """Finds all partitions in the system and creates Partition objects for each one""
        self.findAllPartitions()

        #creates a dictionary mapping each connected drive to a dictionary of partition objects for each partition in the drive
        def findAllPartitions(self):
            self.root = None
            self.primary = None
            self.secondary = None
            self.drives = {}
            self.partitions = {}
            #find all drives in the system
            info = executecommand(["lsblk","-f"]) [0]
            ext_drives = set(re.findall("sd.",info))
            for drive in ext_drives:
                #get a list of all the drive’s partitions
                partitions = []
                s = info.find(drive)
                length = len(drive)
                splice = info[s:length:]
                parts = re.findall(drive+".",splice)
                self.drives[drive] = []
                #create a partition object for each partition
                #checking along the way for partitions mounted as root, prim., sec.
                for part in parts:
                    part = "/dev/"+part
                    partition = Partition(part)
                    self.drives[drive].append(partition)
                    self.partitions[part] = partition
                    if partition.getMountPoint() == "/":
                        self.root = partition
                    elif partition.getMountPoint() == "/opt/data":
"""
self.primary = partition
elif partition.getMountPoint() == "/opt/secondary":
    self.secondary = partition
    #if the drive is not partitioned, create a single
    # partition object for the whole drive
    if len(self.drives[drive]) == 0:
        part = "/dev/"+drive
        partition = Partition(part)
        self.drives[drive].append(partition)
        self.partitions[part] = partition
        if partition.getMountPoint() == "/":
            self.root = partition
        elif partition.getMountPoint() == "/opt/data":
            self.primary = partition
        elif partition.getMountPoint() == "/opt/secondary":
            self.secondary = partition

# checks whether the system has a partition mounted to /opt/data
def hasPrimary(self):
    if self.primary == None:
        return False
    else:
        return True

# checks whether the system has a partition mounted to /opt/secondary
def hasSecondary(self):
    if self.secondary == None:
        return False
    else:
        return True

# unmounts the partition mounted to /opt/data
def unmountPrimary(self, parent):
    if self.primary != None:
        self.primary.unmount(parent)

# unmounts the partition mounted to /opt/secondary
def unmountSecondary(self, parent):
    if self.secondary != None:
        self.secondary.unmount(parent)
#!/usr/bin/env python

# Will not work with multiline entries

class PrettyTable():
    def __init__(self, rows):
        
        Takes in a list of table entries and creates a table using 
        characters such as / and -

        Arguments:
        rows {List of lists} -- Each item in the list is a 
        sub-list of table entries

        
        self.rows = rows
        self.table = None
        self.getalllengths()
        self.draw()

    def setrows(self, rows):
        self.rows.set(rows)

    # creates the table from the row entries
    def draw(self):
        _r_0 = "-"*(self.getlength() + 2*len(self.rows[0]) - 1) + "\n"
        self.table = _r_0
        for row in self.rows:
            r = self.makerow(row)
            self.table += r + "\n"
        self.table += _r_0

    # finds the length of the longest row in the table
    def getlength(self):
        longest = 0
        for row in self.rows:
            length = 0
            for entry in row:
                entry = str(entry)
                length += len(entry)
            if length > longest:
                longest = length
        return longest
#gets the longest entry at the column indicated by index
def getlongest(self,index):
    longest = 0
    for row in self.rows:
        length = len(str(row[index]))
        if length > longest:
            longest = length
    return longest

#creates a list of column lengths
def getalllengths(self):
    self.lengths = []
    for i in range(len(self.rows[0])):
        self.lengths.append(self.getlongest(i))

#draw one row
def makerow(self,row):
    _row_0 = "|
    _row_1 = "|
    for i in range(len(row)):
        entry = str(row[i])
        entry = self.padentry(entry,i)
        _row_0 = _row_0 + entry + "|
        _row_1 = _row_1 + "-"*len(entry)+"+
    _row_1 = _row_1[:-1]+"|
    if row == self.rows[-1]:
        return _row_0
    return _row_0 + "\n" + _row_1

#takes the string indicated by entry and pads it until its
#the length of the longest row at the column indicated by index
def padentry(self,entry,index):
    l = self.lengths[index]
    diff = l - len(entry)
    entry = entry + " "*diff
    return entry
#!/usr/bin/env python
from Tkinter import *
from ttk import Progressbar, Labelframe
from math import floor
from helperfunctions import *
from nilmtab import NilmTab
import partition
import meters
from nilmwidgets import *
from nilmthread import ThreadGroup

class StatusTab(NilmTab):
    def __init__(self, parent, text, master):
        """StatusTab class for NILM Management Interface.

        Arguments:
        parent {Tkinter frame} -- The parent frame of the nilm tab
text {String} -- The text to be displayed on the tab itself
master {Tkinter frame} -- The main frame of the application
""
        NilmTab.__init__(self, parent, text, master)
        self.tg = ThreadGroup([self.update], 1)
        self.addWidgets()
        self.start()
        self.update()

        # update function, called every 0.5 seconds whenever the status tab is selected
    def update(self):
        # reset the system error indicators
        self.SystemError = False
        self.SystemErrorMsg = set()
        # check the new system state and update the tab accordingly
        if self.state == 0:
            self.switch_zero()
        elif self.state == 3:
            self.SystemError = True
```python
self.SystemErrorMsg.add("Warning: Capture has entered a failed state.")
self.switch_two()
elif self.state == 1:
    self.switch_one()
elif self.state == 2:
    self.switch_two()
elif self.state == 4:
    self.switch_four()
elif self.state == 5:
    self.SystemError = True
    self.SystemErrorMsg.add("Warning: Capture has entered a failed state.")
    self.switch_one()

#update the state of the meters
self.checkMeters()
#check cpu and disk use
self.checkCPU()
self.checkDiskUse()
#check if any system errors have been raised and print error messages accordingly
self.checkOverallState()

#called once, when the status tab is first selected
#initializes variables to keep track of detected errors
#finds all meters in the system, creating UI elements for each
def start(self):
    self.SystemError = False  #Set to True if any error is detected in the system
    self.SystemErrorMsg = set()  #A set of all current error messages

    #Frame to hold all meter error message boxes
    self.meterlabelframe = Labelframe(self,text="Meters",width=int(floor(0.475*self.w)))
    self.meterlabelframe.grid(row=1,column=1,sticky="ns")
    self.meterframe = Canvas(self.meterlabelframe,
                              height=int(floor(0.65*self.h)),width=int(floor(0.475*self.w)))
    self.meterframe.grid(row=0,column=0)
    self.meterframe.grid_propagate(False)
    self.meterframe.columnconfigure(0,weight=10)

    #Get all meters returns a dictionary of mapping all meters in the system
    #To their error message, or an empty string if nothing is wrong with the meter
    self.meterErrMsgs = getAllMeters() #Dictionary mapping meters to error msgs
    self.meters = self.meterErrMsgs.keys() #List of all meters

    #Save pointers to UI elements for all the meters
```
self.meterLabels = {}
self.meterErrBoxes = {}
self.meterIndicatorCanvases = {}
self.calbuttons = []
self.nilmscopebuttons = []  # Iterate through all meters and add UI elements for each
i = 1  # i keeps track of the row in which to place the meter frame
self.meters.sort()  # Sort the meters before displaying them
for m in self.meters:
    # Initialize UI elements for the meter
    meterframe = Frame(self.meterframe, height=int(floor(0.1*self.h)), width=int(floor(0.475*self.w)))
    meterframe.columnconfigure(1, weight=1)
    meterframe.grid(row=i)
    self.meterframe.rowconfigure(int(i), weight=10)
    meterframe.grid_propagate(False)
    meterLabel = nilmLabel(meterframe, m, fontsize=18)
    meterLabel.grid(row=1, column=0, sticky="w")
    meterIndicatorCanvas = Canvas(meterframe, width=32, height=22)
    meterIndicatorCanvas.grid(row=1, column=1, sticky="w")
    meter = meters.meter(m, self)
    calbutton = nilmButton(meterframe, text="Calibrate", fontsize=18, command=meter.calibrate)
    calbutton.grid(row=1, column=2, sticky="w")
    nsbutton = nilmButton(meterframe, text="NILM Scope", fontsize=18, command=meter.nilmscope)
    nsbutton.grid(row=1, column=3, sticky="w")
    self.nilmscopebuttons.append(nsbutton)

    if "IP address" in self.meterErrMsgs[m]:
        calbutton.grid_remove()
        if self.meterErrMsgs[m] == "IP address":
            self.meterErrMsgs[m] = ""
        else:
            self.calbuttons.append(calbutton)
    meterErrMsg = Text(meterframe, font=('', 16))
    meterErrMsg.grid(row=2, column=0, sticky="nswe", columnspan=4)
    # Check whether there is an error, and update the error msg
    # And indicator for the corresponding meter
if self.meterErrMsgs[m] == "":  # If there is no error associated with the current meter
    statusIndicator = meterIndicatorCanvas.create_oval(7, 2, 25, 20, fill="dark green")  # Green indicator
    meterErrMsg.insert("1.0", "No error message to display")
else:  # Otherwise, if there is an error associated with the current meter
    statusIndicator = meterIndicatorCanvas.create_oval(7, 2, 25, 20, fill="red3")  # Red indicator
    meterErrMsg.insert("1.0", self.meterErrMsgs[m])

    meterErrMsg.config(state="disabled")  # Disable the error msg text box

    # Save pointers to all UI elements and update i
    self.meterIndicatorCanvases[m] = meterIndicatorCanvas
    self.meterLabels[m] = meterLabel
    self.meterErrBoxes[m] = meterErrMsg
    i += 1

self.buttonframe = Frame(self.meterlabelframe)
self.buttonframe.grid(row=i+2, column=0)

    # configButton provides a shortcut to editing the meters.yml file
    # Meters
    self.configButton = nilmButton(self.buttonframe, text="Configure Meters",
        fontsize=16, height=3, width=20, command=self.meterconfig)
    self.configButton.grid(row=0, column=1)

    # If there are more than 3 meters, add a scroll bar
    if len(self.meters) >= 4:
        self.addMeterScrollBar()

    # adds all remaining widgets to the tab
addWidgets(self):

    # Add status frame to display overall system error messages
    self.overview0 = Labelframe(self, text="Overview")
    self.overview0.grid(row=1, column=0)

    self.overview = Frame(self.overview0)
    self.overview.grid()
```python
self.stxtframe = Frame(self.statusframe,
    height=int(floor(0.35*self.h)),width=int(floor(0.475*self.w)))
self.stxtframe.grid(sticky="nw")
self.stxtframe.grid_propagate(False)

smolframe = Frame(self.stxtframe)
smolframe.grid(row=0,column=0,sticky="w")

self.statustext = nilmLabel(smolframe,"System status",fontsize=18)
self.statustext.grid(row=0,column=0,sticky="nw")

self.statusIndicatorCanvas = Canvas(smolframe, width=32, height=22)
self.statusIndicatorCanvas.grid(row=0,column=1,sticky="nw")
self.statusIndicator = self.statusIndicatorCanvas.create_oval(7, 2, 25, 20, fill="dark green")

self.erroroutput = Text(self.stxtframe,height=15,font="",16),width=70)
self.erroroutput.grid(row=1,column=0,columnspan=2,sticky="nw")
self.erroroutput.insert("1.0","No error messages to display")
self.erroroutput.config(state="disabled")

#Add progressbars and labels to display cpu and disk use
self.cpuframe = Frame(self.statusframe,
    height=int(floor(0.12*self.h)),width=int(floor(0.475*self.w)))
self.cpuframe.grid(row=2,column=0,sticky="nw")
self.cpuframe.grid_propagate(False)

self.cpulabel = nilmLabel(self.cpuframe,"CPU use")
self.cpulabel.grid(row=0,column=0,sticky="w")
self.cpulabel.config(font="",18))

self.numCores = getNumCores()
self.cpuBars = {}
self.cpuLabels = {}
for i in range(self.numCores):
    cpulabell = nilmLabel(self.cpuframe,"Core "+str(i+1))
    cpulabell.grid(row=1+i,column=0)
    cpulabell.config(font="",18))

    cpuBar = Progressbar(self.cpuframe,orient="horizontal",
    length=int(floor(0.3*self.w)),mode="determinate",maximum=100,value=30)
    self.cpuBars[i+1] = cpuBar
    cpuBar.grid(row=1+i,column=1,sticky="w")

    cpuLabel2 = nilmLabel(self.cpuframe, "30%")
    cpuLabel2.grid(row=1+i,column=2)
    cpuLabel2.config(font="",18))
```
self.cpuLabels[i+1] = cpulabel2

self.duframe = Frame(self.statusframe, height=int(floor(0.08*self.h)), width=int(floor(0.475*self.w)))
self.duframe.grid(row=3+i,column=0,sticky="n")
self.duframe.grid_propagate(False)

self.dulabel = nilmLabel(self.duframe,"Disk use")
self.dulabel.grid(row=0,column=0,sticky="nw")
self.dulabel.config(font=('',18))

self.diskUse = Progressbar(self.duframe,orient="horizontal", length=int(floor(0.3*self.w)),mode="determinate",maximum=100,value=50)
self.diskUse.grid(row=1,column=1,sticky="nw")
self.diskUse.grid(row=3+i,column=0,sticky="n")

self.dulabel2 = nilmLabel(self.duframe,"50"+"%")
self.dulabel2.grid(row=1,column=2)
self.dulabel2.config(font=('',18))

#Adds a scrollbar to the meter frame when there are more than 5 meters
def addMeterScrollBar(self):
    self.meterscroll = Scrollbar(self.meterlabelframe, orient=VERTICAL, command=self.meterframe.yview)
    self.meterframe.config(yscrollcommand=self.meterscroll.set)
    self.meterscroll.grid(row=0,column=2,sticky="ns")

#Checks for errors with the meters
def checkMeters(self):
    #getAllMeters returns a dictionary mapping each meter to any error messages for it
    self.meterErrMsgs = getAllMeters()
    self.meters = self.meterErrMsgs.keys()
    self.meters.sort()

    #Check if yml formatted correctly: if it isn’t, all meters throw error
    ymlError = ymlOK()
    if ymlError != "": #if there is an error with the meters file
        for meter in self.meters:
            self.meterErrBoxes[meter].config(state="normal")
            self.meterErrBoxes[meter].delete("1.0",END)
            self.meterErrBoxes[meter].insert("1.0","Syntax error in Meters.yml file:" + "\n" + ymlError)
            self.meterErrBoxes[meter].config(state="disabled")
            self.SystemError = True
            self.SystemErrorMsg.add("Error: Improper syntax in Meters.yml file:" + "\n" + ymlError)
        for meter in self.meters:
self.meterIndicatorCanvases[meter].create_oval(7, 2, 25, 20, fill="red")

# Iterate through the meters, checking for error messages and displaying them
for meter in self.meters:
    if self.meterErrMsgs[meter] == ": No error message to display"
        self.meterErrBoxes[meter].config(state="disabled")
    else:
        self.meterErrBoxes[meter].config(state="normal")
        self.meterErrBoxes[meter].delete("1.0", END)
        self.meterErrBoxes[meter].insert("1.0", self.meterErrMsgs[meter])

# Gets the CPU use for each core and generates an error message if any are high
def checkCPU(self):
    cpuUse = getCPUuse(self.numCores)
    for c in cpuUse.keys():
        self.cpuBars[c].config(value=cpuUse[c])
        self.cpuLabels[c].config(text=str(cpuUse[c]) + ": "+str(c))
        if cpuUse[c] > 80:
            self.SystemError = True
            self.SystemErrorMsg.add("Warning: CPU use high for core "+str(c))

# Gets the disk use of the mounted drive and generates an error message if its high
def checkDiskUse(self):
    diskUse = getDiskUse("/opt/data")
    self.diskUse.config(value=float(diskUse[-1]))
    self.dulabel2.config(text=str(diskUse))
    if float(diskUse[-1]) > 80:
        self.SystemError = True
        self.SystemErrorMsg.add("Warning: Disk is almost full. Consider switching it out soon.")

# Checks if any system errors have been raised, and displays them if so
def checkOverallState(self):
    self.erroroutput.config(state="normal")
```python
self.erroroutput.delete("1.0", END)
if self.SystemError == True:
    errorMsg = ""
    for error in self.SystemErrorMsg:
        self.statusIndicator =
        self.statusIndicatorCanvas.create_oval(7, 2, 25, 20, fill="red3")
        errorMsg = errorMsg + error + "\n"
else:
    self.statusIndicator = self.statusIndicatorCanvas.create_oval(7, 2, 25, 20, fill="dark green")
    self.errorMsg = "No error messages to display"
self.erroroutput.insert("1.0", errorMsg)
self.erroroutput.config(state="disabled")

# pulls up the meters.yml file to allow the user to configure it
def meterconfig(self):
    configMeters(self)
    self.start()  # force redraw

# the switch functions update the state of various buttons depending on
# whether the capture and
def switch_zero(self):
    self.configButton["state"] = "disabled"
    for b in self.calbuttons:
        b["state"] = "disabled"
    for b in self.nilmscopebuttons:
        b["state"] = "disabled"

def switch_one(self):
    self.configButton["state"] = "normal"
    for b in self.calbuttons:
        b["state"] = "normal"
    for b in self.nilmscopebuttons:
        b["state"] = "normal"

def switch_two(self):
    self.configButton["state"] = "disabled"
    for b in self.calbuttons:
        b["state"] = "disabled"
    for b in self.nilmscopebuttons:
        b["state"] = "disabled"

def switch_four(self):  # throws a system error; database should not be
    self.configButton["state"] = "disabled"
    self.SystemError = True
    self.SystemErrorMsg.add("Warning: Capture is running but database is
    stopped.")
    for b in self.calbuttons:
        b["state"] = "disabled"
```
for b in self.nilmscopebuttons:
b["state"] = "disabled"
C.3 Load Identifier

Source Code C.22

Filename: stbd-detect-classify.py

Collaborators: Daisy H. Green

Description: Detects and classifies loads. Constants (not shown) and classification logic has been tuned to loads on the Starboard subpanel aboard a USCG Cutter. Variants of this algorithm would be able to detect a different set of loads.

```python
from joule.client import FilterModule
import asyncio
import numpy as np
import pandas as pd
from scipy import signal

class Stbd_Detect_and_Classify(FilterModule):
    # Detects and classifies events on the stbd panel
    def __init__(self):
        self.stop_requested = False
        self.description = "detects events on stbd panel"
        self.help = "help: detects events on stbd panel"
        self.min_values_on = self.matlab_to_python('<MATLAB OMITTED>')
        self.max_values_on = self.matlab_to_python('<MATLAB OMITTED>')
        self.min_values_off = self.matlab_to_python('<MATLAB OMITTED>')
        self.max_values_off = self.matlab_to_python('<MATLAB OMITTED>')
        self.b2 = self.matlab_to_python('<MATLAB OMITTED>')
        self.b3 = self.matlab_to_python('<MATLAB OMITTED>')
        self.b4 = self.matlab_to_python('<MATLAB OMITTED>')
        self.W2 = self.matlab_to_python('<MATLAB OMITTED>')
        self.W3 = self.matlab_to_python('<MATLAB OMITTED>')
        self.W4 = self.matlab_to_python('<MATLAB OMITTED>')
        self.b2_off = self.matlab_to_python('<MATLAB OMITTED>')
        self.b3_off = self.matlab_to_python('<MATLAB OMITTED>')
        self.b4_off = self.matlab_to_python('<MATLAB OMITTED>')
        self.W2_off = self.matlab_to_python('<MATLAB OMITTED>')
        self.W3_off = self.matlab_to_python('<MATLAB OMITTED>')
        self.W4_off = self.matlab_to_python('<MATLAB OMITTED>')

    def custom_args(self, parser):
        pass

    def rollingmedian(self, data, window_size):
```

253
return pd.Series(data).rolling(window=window_size, center=True, min_periods=1).median().values

def movingaverage(self, data, window_size):
    window = np.ones(int(window_size)) / float(window_size)
    avg = np.convolve(data, window, 'same')
    return avg

def softmax(self, x):
    return np.exp(x) / np.sum(np.exp(x), axis=0)

def matlab_to_python(self, mat):
    return np.array(np.matrix(mat.strip('[]')))

async def run(self, parsed_args, inputs, outputs):
    stream_in = inputs['input']
    stream_out = outputs['output']
    while not self.stop_requested:
        sarray_in = await stream_in.read()
        time = sarray_in['timestamp']
        data_in = sarray_in['data']

        # minimum length of data block
        if (len(sarray_in) < 600):
            await asyncio.sleep(1)
            continue

        # length of data block
        array_length = len(sarray_in)

        PA = data_in[:, 0]
        PB = data_in[:, 8]
        PC = data_in[:, 16]
        QA = data_in[:, 1]
        QB = data_in[:, 9]
        QC = data_in[:, 17]

        WINDOW_SIZE = 101

        # median filter to smooth
        PA_filt = self.rollingmedian(PA, WINDOW_SIZE)
        QA_filt = self.rollingmedian(QA, WINDOW_SIZE)
        PB_filt = self.rollingmedian(PB, WINDOW_SIZE)
        QB_filt = self.rollingmedian(QB, WINDOW_SIZE)
        PC_filt = self.rollingmedian(PC, WINDOW_SIZE)
        QC_filt = self.rollingmedian(QC, WINDOW_SIZE)
        SB_filt = (PB_filt**2 + QB_filt**2)**(0.5)
PQ_filt = np.hstack((PA_filt[:,None], QA_filt[:,None], PB_filt[:,None],
                      QB_filt[:,None], PC_filt[:,None], QC_filt[:,None]))

PQ = np.hstack((PA[:,None], QA[:,None], PB[:,None],
                QB[:,None], PC[:,None], QC[:,None]))

f_line = 60

# threshold for minimum P_total or Q_total
min_on = 1000

# window width for steady-state change calculations
ss_med = 0.5  # seconds
ss_med = int(round(ss_med*f_line))  # in indices

# for finding end of on-transient
dP = np.diff(PB)
dP = np.append(dP, [0])
average = self.movingaverage(dP, 3)

# find indices for on and off transients
# smoothing with gaussian
kernal = np.diff(np.diff(signal.gaussian(60, std=5)))
h = (np.convolve(kernal, SB_filt, 'same'))

# setting anything less than threshold to zero
h[np.where(abs(h) < 10)] = 0

# finding zero-crossings
# sign change is two
ind2 = np.where(np.abs(np.diff(np.sign(h))) == 2)[0] - 1

# two sign changes of one in a row
ind1 = np.where(np.diff(np.sign(h)))[0]
ind1 = ind1[np.where(np.diff(ind1) == 1)]

# sorting indices
ind_locations = np.sort(np.append(ind1, ind2))
num_of_ind = len(ind_locations)  # total number of indices

# sign of h determines if on-event(1) or off-event(-1)
ind = np.sign(h[ind_locations])

# pre-allocating
time_new = np.zeros(num_of_ind, dtype=time.dtype)
index_on = np.zeros(num_of_ind)
index_off = np.zeros(num_of_ind)
data_ss = np.zeros((num_of_ind, 6), dtype=float)
data_peak = np.zeros((1, 6), dtype=float)
for y in range(num_of_ind):
    x = int(ind_locations[y])
    if (x > 60) and x < (array_length - 60):
        if ind[y] == 1:
            vector = average[x:x+180]
            flat = np.where(abs(vector) < 10)
            flat = flat[0]
            flat = flat[0]

            ind_start = x - ss_med
            ind_end = x + flat

            if ind_start < 0:
                ind_start = 0

            time_new[y] = time[x]

    def find_ss_on(pstream,ind_start,ind_end,ss_med,pos):
        return np.median(pstream[ind_end:ind_end + ss_med]) - np.median(pstream[ind_start:pos - 1])

    def find_peak(pstream,ind_start,ind_end,ss_med,pos):
        return np.amax(pstream[ind_start:ind_end]) - np.median(pstream[ind_end:ind_end+ss_med])

    for i in range(6):
        data_ss[y,i] = find_ss_on(PQ_filt[:,i],ind_start,ind_end,ss_med,x)
        data_peak[0,i] = find_peak(PQ[:,i],ind_start,ind_end,ss_med,x)

ON = np.hstack((data_ss[y,0:2],data_peak[0,0:2],data_ss[y,2:4],data_peak[0,2:4],
                data_ss[y,4:6],data_peak[0,4:6]))
ON_norm = np.divide(np.subtract(ON ,
                          self.min_values_on),np.subtract(self.max_values_on ,
                          self.min_values_on));

Z2 = np.dot(ON_norm,self.W2) + self.b2
A2 = np.maximum(0,Z2)
Z3 = np.dot(A2,self.W3) + self.b3
A3 = np.maximum(0,Z3)
Z4 = np.dot(A3,self.W4) + self.b4
Z4 = np.transpose(Z4)
A4 = self.softmax(Z4)
value_on = max(A4)

# Threshold for confidence in classifier
if value_on > 0.2:
    index_on[y] = np.argmax(A4) + 1
else:
    index_on[y] = 0

if ind[y] == -1:
    ind_start = x - ss_med
    ind_end = x + 1

    if ind_start < 0:
        ind_start = 0

    #time_new[0,y] = time[x]
    time_new[y] = time[x]

    def find_ss_off(pstream,ind_start,ind_end,ss_med,pos):
        return np.median(pstream[ind_end:ind_end + ss_med]) - np.median(pstream[ind_start:pos - 1])

    for i in range(6):
        data_ss[y,i] = find_ss_off(PQ_filt[:,i],ind_start,ind_end,ss_med,x)

        OFF = data_ss[y,:]
        OFF_norm = np.divide(np.subtract(OFF ,
                                   self.min_values_off),np.subtract(self.max_values_off ,
                                   self.min_values_off));

        Z2 = np.dot(OFF_norm,self.W2_off) + self.b2_off
        A2 = np.maximum(0,Z2)
        Z3 = np.dot(A2,self.W3_off) + self.b3_off
        A3 = np.maximum(0,Z3)
        Z4 = np.dot(A3,self.W4_off) + self.b4_off
        Z4 = np.transpose(Z4)
        A4 = self.softmax(Z4)
        value_off = max(A4)

        # Threshold for confidence in classifier#
        if value_off > 0.2:
            index_off[y] = np.argmax(A4) + 1
        else:
            index_off[y] = 0

        P_total = data_ss[:,0] + data_ss[:,2] + data_ss[:,4]
        Q_total = data_ss[:,1] + data_ss[:,3] + data_ss[:,5]
# only keep if the steady-state is greater than the min threshold

good_ind = np.where( (abs(P_total) > min_on) | (abs(Q_total) > min_on) )
num_of_ind = len(good_ind[0])

P_total = P_total[good_ind]
Q_total = Q_total[good_ind]
index_on = index_on[good_ind]
index_off = index_off[good_ind]
time_new = time_new[good_ind]

print(P_total)
print(Q_total)

# Classifying transients #
# Load 1: MDE LO Heater
# Load 2: MDE JW Heater
# Load 3: MDE PL Pump
# Load 4: SSDG JW Heater
# Load 5: SSDG LO Heater
# Load 6: CPP Pump
# Load 7: Oily Water Separators

if num_of_ind > 0:

    EVENT_ON = 1
    EVENT_OFF = 2

    loads = [None]  # pad with None for 1-indexed list

    for i in range(1,8):
        loads.append(np.zeros(num_of_ind))
        loads[i][index_on==i] = EVENT_ON
        loads[i][index_off==i] = EVENT_OFF

    loads[1][index_off==9] = EVENT_OFF
    loads[2][index_off==9] = EVENT_OFF

    loads[1][index_on==10] = EVENT_ON
    loads[3][index_on==10] = EVENT_ON

    loads[2][index_on==11] = EVENT_ON
    loads[3][index_on==11] = EVENT_ON

    for i in range(1,4):
        loads[i][index_on==12] = EVENT_ON
        loads[i][index_off==9] = EVENT_OFF

    loads[1][index_off==8] = EVENT_OFF
loads[2][index_off==8] = EVENT_OFF

# Putting P and Q Values into output for each load #
pow_P, pow_Q = [None], [None]  # pad with None for 1-indexed

list

for i in range(1,8):
    pow_P.append(np.zeros(num_of_ind))
    pow_Q.append(np.zeros(num_of_ind))
    pow_P[i][index_on==i] = P_total[index_on==i]
    pow_Q[i][index_on==i] = Q_total[index_on==i]
    pow_P[i][index_off==i] = P_total[index_off==i]
    pow_Q[i][index_off==i] = Q_total[index_off==i]

pow_P[7][index_on==8] = P_total[index_on==8]
pow_P[7][index_on==8] = Q_total[index_on==8]

# Adjusting for when MDE loads turn on simultaneously
pow_P[1][index_on==9] = (P_total[index_on==9])*0.56
pow_Q[1][index_on==9] = (Q_total[index_on==9])*0.51
pow_P[2][index_on==9] = (P_total[index_on==9])*0.44
pow_Q[2][index_on==9] = (Q_total[index_on==9])*0.49

pow_P[1][index_on==10] = (P_total[index_on==10])*0.92
pow_Q[1][index_on==10] = (Q_total[index_on==10])*0.13
pow_P[3][index_on==10] = (P_total[index_on==10])*0.08
pow_Q[3][index_on==10] = (Q_total[index_on==10])*0.87

pow_P[2][index_on==11] = (P_total[index_on==11])*0.90
pow_Q[2][index_on==11] = (Q_total[index_on==11])*0.12
pow_P[3][index_on==11] = (P_total[index_on==11])*0.10
pow_Q[3][index_on==11] = (Q_total[index_on==11])*0.88

pow_P[1][index_on==12] = (P_total[index_on==12])*0.53
pow_Q[1][index_on==12] = (Q_total[index_on==12])*0.11
pow_P[2][index_on==12] = (P_total[index_on==12])*0.42
pow_Q[2][index_on==12] = (Q_total[index_on==12])*0.10
pow_P[3][index_on==12] = (P_total[index_on==12])*0.06
pow_Q[3][index_on==12] = (Q_total[index_on==12])*0.78

# Adjusting for when MDE loads turn off simultaneously
pow_P[1][index_off==8] = (P_total[index_off==8])*0.59
pow_Q[1][index_off==8] = (Q_total[index_off==8])*0.73
pow_P[2][index_off==8] = (P_total[index_off==8])*0.41
pow_Q[2][index_off==8] = (Q_total[index_off==8])*0.27

pow_P[1][index_off==9] = (P_total[index_off==9])*0.57
pow_Q[1][index_off==9] = (Q_total[index_off==9])*0.12
pow_P[2][index_off==9] = (P_total[index_off==9])*0.39
pow_Q[2][index_off==9] = (Q_total[index_off==9])*0.04
pow_P[3][index_off==9] = (P_total[index_off==9])*0.04
pow_Q[3][index_off==9] = (Q_total[index_off==9])*0.84
output_len = num_of_ind

sarray_out = np.zeros(output_len, dtype=stream_out.dtype)
cols = []
for i in range(1,8):
    cols.extend([loads[i][:,None], pow_P[i][:,None],
                  pow_Q[i][:,None]])
data_out = np.hstack(cols)
sarray_out['timestamp'] = time_new
sarray_out['data'] = data_out

    await stream_out.write(sarray_out)

stream_in.consume(len(sarray_in)-120)

    def stop(self):
        self.stop_requested = True

if __name__ == "__main__":
    r = Stbd_Detect_and_Classify()
    r.start()
from joule.client import FilterModule
import asyncio
import numpy as np

from joule.utils import nilmdb

class Event_Confirm(FilterModule):
    " Confirms events on the stbd panel with FSM"

    def __init__(self):
        #super(Event_Confirm, self).__init__("Event Confirm")
        self.stop_requested = False
        self.description = "confirms events with FSM"
        self.help = "help: confirms events with FSM"
        self.previous_state = 2
        self.previous_time = 0

    def custom_args(self, parser):
        parser.add_argument("index", type=int,
                            help="index of load")
        parser.add_argument("path", type = str)

    async def run(self, parsed_args, inputs, outputs):
        stream_in = inputs["input"]
        stream_out = outputs["output"]

        while(1):
            sarray_in = await stream_in.read()
            time = sarray_in['timestamp']
            data_in = sarray_in['data']

            N = parsed_args.index
            state_ind = N + 2*N
            P_ind = N + 1 + 2*N
            Q_ind = N + 2 + 2*N

            event_state = data_in[:,state_ind]
            real_power = data_in[:,P_ind]
            reactive_power = data_in[:,Q_ind]

            ind = np.nonzero(event_state)
            num_of_ind = np.count_nonzero(event_state)
event_total = event_state[ind]
event_time = time[ind]
event_real_power = real_power[ind]
event_reactive_power = reactive_power[ind]

new_time = []
state_out = []
P_out = []
Q_out = []

for x,y in enumerate(event_total):
    # if first data point in interval, have to check previous state
    if x == 0:
        # if first data point is ON
        if y == 1:
            # if previous state is ON, invalid so delete data point at previous state
            if self.previous_state == 1:
                client = nilmdb.AsyncClient("http://nilm.primary/nilmdb")
                await client.stream_remove(parsed_args.path,
                                           self.previous_time, self.previous_time+1)
                client.close()

    # if there is only one data point in interval,
    output current ON
    if num_of_ind == 1:
        new_time.append(event_time[x])
        state_out.append(1)
        P_out.append(event_real_power[x])
        Q_out.append(event_reactive_power[x])

    # if there is more than one data point in interval,
    # check that the next point is an OFF
    # only output if ON followed by OFF
    if num_of_ind > 1:
        if event_total[x+1] == 2:
            new_time.append(event_time[x])
            new_time.append(event_time[x+1])
            state_out.append(1)
            state_out.append(2)
            P_out.append(event_real_power[x])
            P_out.append(event_real_power[x+1])
            Q_out.append(event_reactive_power[x])
            Q_out.append(event_reactive_power[x+1])

    # if previous state is an OFF
    if self.previous_state == 2:
# if there is only one data point in interval,
if num_of_ind == 1:
    new_time.append(event_time[x])
    state_out.append(1)
    P_out.append(event_real_power[x])
    Q_out.append(event_reactive_power[x])

# if there is more than one data point in
# interval,
# check that the next point is an OFF
# only output if ON followed by OFF
if num_of_ind > 1:
    if event_total[x+1] == 2:
        new_time.append(event_time[x])
        new_time.append(event_time[x+1])
        state_out.append(1)
        state_out.append(2)
        P_out.append(event_real_power[x])
        P_out.append(event_real_power[x+1])
        Q_out.append(event_reactive_power[x])
        Q_out.append(event_reactive_power[x+1])

# if first data point is an OFF
if y == 2:
    # if previous state is an ON, then output current
    if self.previous_state == 1:
        new_time.append(event_time[x])
        state_out.append(2)
        P_out.append(event_real_power[x])
        Q_out.append(event_reactive_power[x])

# if not first data point
if x > 0:
    # if ON detected
    if y == 1:
        # if last data point in interval, output ON
        if x == (num_of_ind -1):
            new_time.append(event_time[x])
            state_out.append(1)
            P_out.append(event_real_power[x])
            Q_out.append(event_reactive_power[x])
        # if NOT last data point in interval, only output if
        if x < (num_of_ind) -1:
            if event_total[x+1] == 2:
                new_time.append(event_time[x])
                new_time.append(event_time[x+1])
                state_out.append(1)
                state_out.append(2)
```
output_len = len(state_out)
print(output_len)
if output_len > 0:
    # saving last state of load
    self.previous_state = state_out[-1]
    self.previous_time = new_time[-1]
    sarray_out = np.zeros(output_len, dtype=stream_out.dtype)
    state_out = np.array(state_out)
    P_out = np.array(P_out)
    Q_out = np.array(Q_out)

    data_out = np.hstack((state_out[:, None], P_out[:, None], Q_out[:, None]))
    sarray_out['timestamp'] = new_time
    sarray_out['data'] = data_out

    await stream_out.write(sarray_out)
    stream_in.consume(len(sarray_in))

    # def stop(self):
    #     self.stop_requested = True

if __name__ == "__main__":
    r = Event_Confirm()
    r.start()
```
from joule.utils.time import now as time_now
from joule.client import ReaderModule
import asyncio
import numpy as np
import logging
import yaml
import shlex, subprocess

appliance_file = "/opt/load-detector/appliances.yml"

class RealtimeReader(ReaderModule):
    def __init__(self):
        #super(RealtimeReader, self).__init__()
        self.output_rate = 1 # one chunk per second
        self.stop_requested = False
        self.description = "import data"
        self.help = "importdata"
        self.elapsed_samples = 0

        with open(appliance_file, 'r') as f:
            self.appliances = yaml.load(f)

        settings = self.appliances["reader"]
        self.settings = settings
        self.paths = settings["paths"]
        self.unified = settings["unified"]

        self.files = [open(p, 'r') for p in self.paths]
        self.rate = settings["rate"] if 'rate' in settings else 60
        self.mode = settings["mode"] if 'mode' in settings else "live"

    def run_cmd(s):
        cmd = shlex.split(s)
        p = subprocess.Popen(cmd, stdout=subprocess.PIPE)
        return p.communicate()[0]

        first_line = run_cmd("head -1 %s" % self.paths[0]))
        last_line = run_cmd("tail -1 %s" % self.paths[0]))
        if last_line == "":
            last_line = run_cmd("tail -2 %s" % self.paths[0]))
        ts_first = int(first_line.decode().split(" ")[0].strip())
```python
ts_last = int(last_line.decode().split(" ")[0].strip())
delta = ts_last - ts_first

self.ts_first, self.ts_last = ts_first, ts_last

self.timestamp = float(time_now()) if self.mode == "live" else float(self.ts_first)
self.time_start = self.timestamp

def custom_args(self, parser):
    pass

async def run(self, parsed_args, output):
    mode = self.mode
    data_ts_inc = 1/self.rate*1e6
    self.output_period = 1/self.output_rate
    output_block = int(round(self.rate/self.output_rate))
    BLOCK_SIZE = output_block if mode == "live" else 10000

    while(not self.stop_requested):
        data = np.zeros((BLOCK_SIZE, 25), dtype=float)

        def read_data(fp, phase, unified):
            count_read = 0
            for i in range(BLOCK_SIZE):
                line = fp.readline()
                if line == "":  # loop file
                    fp.seek(0)
                else:
                    break
            count_read += 1
            vec = np.fromstring(line, dtype=float, sep=' ')
            if phase == 0:
                data[i, 0] = vec[0]  # copy timestamp always
            if unified:
                data[i,1:25] = vec[1:25]
            else:
                data[i,phase*8+1:phase*8+9] = vec[1:9]
            return count_read

        if not self.unified:
            for ph in range(3):
                count = read_data(self.files[ph], ph, False)
        else:
            count = read_data(self.files[0], 0, True)

        if mode == "live":  # generate timestamps
            for i in range(BLOCK_SIZE):
                data[i,0] = int(self.timestamp)
```

self.timestamp += data_ts_inc

await output.write(data[:count])

if mode == "live":
    # calculate adjusted wait time
    elapsed_time = (time_now() - self.time_start) / 1.0e6
    self.elapsed_samples += count
    estimated_time = self.elapsed_samples / float(self.rate)
    time_delta = estimated_time - elapsed_time
    if time_delta < -self.output_period:
        wait_time = 0  # skip wait, catch up
    else:
        wait_time = self.output_period + time_delta

    await asyncio.sleep(wait_time)
else:
    if self.timestamp > self.ts_last:
        await asyncio.sleep(10000)  # just spool
    else:
        pass  # keep going without waiting

if __name__ == "__main__":
    r = RealtimeReader()
    r.start()
Filename: appliances.yml
Collaborators: Daisy H. Green
Description: Configuration file including loads from the Starboard panel, specifying to run the Starboard detector file. The reader is circularly injecting data from a PREP file.

```yaml
###
# This file lists the appliances for this NILM
#

## STBD PANEL ##

reader:
  enabled: true
  unified: true
  filter: /RealtimeReader.py
  paths:
    - /home/nilm/data/stbd3-unified.dat
  output: /spencer_stbd/prep

stbd:
  enable: true
  filter: /stbd_detect_and_classify.py
  input: /spencer_stbd/prep
  main_path: /spencer_stbd/
  settings:
    datatype: float32
    keep: 1y
    decimate: yes
  loads:
    appliance1:
      enabled: true
      name: mde_lo
      index: 0
    appliance2:
      enabled: true
      name: mde_jw
      index: 1
    appliance3:
      enabled: true
      name: mde_prelube
      index: 2
```

268
appliance4:
    enabled: true  # true or false
    name: ssdg_jw  # name of appliance
    index: 3       # index of appliance

appliance5:
    enabled: true  # true or false
    name: ssdg_lo  # name of appliance
    index: 4       # index of appliance

appliance6:
    enabled: true  # true or false
    name: cpp      # name of appliance
    index: 5       # index of appliance

appliance7:
    enabled: true  # true or false
    name: ows      # name of appliance
    index: 6       # index of appliance
Source Code C.26

Filename: load-appliances.py

Collaborators: Daisy H. Green

Description: Sets up the Joule processing framework by creating module and stream specifications to run the filters in the appliances.yml configuration.

```python
#!/usr/bin/python3

import configparser
import os
import sys
import yaml
# from appliances import *

module_dir = '/etc/joule/module_configs'
stream_dir = '/etc/joule/stream_configs'
filter_dir = '/opt/load-detector'
config_file = '/opt/load-detector/appliances.yml'

def load_appliances(appliance_file):
    with open(appliance_file, 'r') as f:
        appliances = yaml.load(f)
    return appliances

def build_module_configs(my_appliance,panel_name,my_panel):
    # sys.stdout.write("generating module configuration ")
    name = my_appliance['name']
    panel = panel_name
    index = my_appliance['index']
    main_path = my_panel['main_path']
    input_path = "%sevents" % (main_path)
    output_path = "%s%s" % (main_path,name)

    # --meter capture configuration--
    config = configparser.ConfigParser()
    config['Main'] = {
        "name": "%s %s " % (panel, name),
        "exec_cmd": "python3 %s/event_confirm.py %s %s %s %s %s" % (filter_dir,index, output_path)}
    config['Inputs'] = {
        "input": input_path}
    config['Outputs'] = {
        "output": output_path}
    #config['Index'] = {"load_index": index}
```
    path = os.path.join(module_dir, "%s_%s.conf" % (panel, name))
    with open(path, 'w+') as configfile:
        config.write(configfile)

def build_stream_configs(my_appliance,panel_name,my_panel):
    #sys.stdout.write("generating stream configuration ")
    name = my_appliance['name']
    panel = panel_name
    main_path = my_panel['main_path']
    output_path = "%s%s" % (main_path,name)
    settings = my_panel['settings']
    config = configparser.ConfigParser()
    config['Main'] = {"name": "%s %s " % (panel, name),
                      "datatype": settings["datatype"],
                      "keep": settings["keep"],
                      "decimate": settings["decimate"],
                      "path": output_path}
    config['Element1'] = {"name": "%s %s " %name,
                          "units": "none",
                          "discrete": "yes"}
    config['Element2'] = {"name": "P",
                          "units": "W",
                          "discrete": "yes"}
    config['Element3'] = {"name": "Q",
                          "units": "W",
                          "discrete": "yes"}

    path = os.path.join(stream_dir, "%s_%s.conf" % (panel, name))
    with open(path, 'w+') as configfile:
        config.write(configfile)

def remove_module_configs(my_appliance,panel_name):
    #sys.stdout.write("removing module configuration ")
    name = my_appliance['name']

    path = os.path.join(module_dir, "%s_%s.conf" % (panel_name, name))
    try:
        os.remove(path)
    except OSError:
        pass

def remove_stream_configs(my_appliance,panel_name):
    #sys.stdout.write("removing stream configuration ")
    name = my_appliance['name']
    path = os.path.join(stream_dir, "%s_%s.conf" % (panel_name, name))
try:
    os.remove(path)
except OSError:
    pass

def build_filter_module_configs(panel_name, my_panel):
    #sys.stdout.write("generating module configuration ")

    filter_path = my_panel['filter']
    input_path = my_panel['input']
    main_path = my_panel['main_path']
    output_path = "\sevents" % (main_path)

    # --meter capture configuration--
    config = configparser.ConfigParser()
    config['Main'] = {
        "name": "%s detect and classify" % (panel_name),
        "exec_cmd": "python3 %s" % (filter_dir, filter_path)
    }
    config['Inputs'] = {
        "input": input_path
    }
    config['Outputs'] = {
        "output": output_path
    }
    path = os.path.join(module_dir, "%s_detect_and_classify.conf" % panel_name)
    with open(path, 'w+') as configfile:
        config.write(configfile)

def build_filter_stream_configs(panel_name, my_panel):
    #sys.stdout.write("generating stream configuration ")
    main_path = my_panel['main_path']
    stream_path = "\sevents" % (main_path)
    settings = my_panel['settings']

    config = configparser.ConfigParser()
    config['Main'] = {
        "name": "%s detect and classify " % (panel_name),
        "datatype": "float32",
        "keep": settings["keep"],
        "decimate": settings["decimate"],
        "path": stream_path
    }

    l = list()

    loads = my_panel["loads"]

    for key, my_appliance in loads.items():
        ind = my_appliance['index'] + 1
        name = my_appliance['name']

        temp = [ind, name]
        l.append(temp)
sorted_list = sorted(l)
num_of_loads = len(sorted_list)

for x, y in enumerate(sorted_list):
    ind = y[0]
    name = y[1]
    config['Element%s %d'] = "name": "%s " %name,
                           "units": "none",
                           "discrete": "yes"
    config['Element%s %d'] = "name": "%s_P" %name,
                           "units": "W",
                           "discrete": "yes"
    config['Element%s %d'] = "name": "%s_Q" %name,
                           "units": "W",
                           "discrete": "yes"

path = os.path.join(stream_dir, "%s_detect_and_classify.conf" % (panel_name))
with open(path, 'w+') as configfile:
    config.write(configfile)

def remove_filter_module_configs(panel_name):
    #sys.stdout.write("removing module configuration ")
    path = os.path.join(module_dir, "%s_detect_and_classify.conf" % (panel_name))
    try:
        os.remove(path)
    except OSError:
        pass

def remove_filter_stream_configs(panel_name):
    #sys.stdout.write("removing stream configuration ")
    path = os.path.join(stream_dir, "%s_detect_and_classify.conf" % (panel_name))
    try:
        os.remove(path)
    except OSError:
        pass

def build_reader_module_configs(my_reader):
    #sys.stdout.write("generating module configuration ")
reader_path = my_panel['filter']
output_path = my_panel['output']

# --meter capture configuration--
config = configparser.ConfigParser()
config['Main'] = {'name': 'realtime reader',
                 'exec_cmd': 'python3 %s%s %'
                 % (filter_dir, reader_path)}

config['Inputs'] = {}
config['Outputs'] = {'output': output_path}

path = os.path.join(module_dir, 'realtime_reader_module.conf')
with open(path, 'w+') as configfile:
    config.write(configfile)

def build_reader_stream_configs(my_reader):

    reader_path = my_panel['filter']
    output_path = my_panel['output']

    config = configparser.ConfigParser()
    config['Main'] = {'name': 'realtime reader',
                     'datatype': 'float32',
                     'keep': '1y',
                     'decimate': 'yes',
                     'path': output_path}

    for i in range(24):
        prefix = 'P' if i % 2 == 0 else 'Q'
        phase = chr(65 + int(i / 8))
        harm = int((i % 8) / 2) * 2 + 1
        name = '%s%s%i' % (prefix, phase, harm)
        config['Element%i' % (i+1)] = {'name': name, 'units': 'W'}

    path = os.path.join(stream_dir, 'realtime_reader_stream.conf')
    with open(path, 'w+') as configfile:
        config.write(configfile)

def remove_reader_module_configs():
    #sys.stdout.write("removing module configuration ")

    path = os.path.join(module_dir, 'realtime_reader_module.conf')
    try:
        os.remove(path)
    except OSError:
        pass

def remove_reader_stream_configs():
    #sys.stdout.write("removing stream configuration ")

    path = os.path.join(stream_dir, 'realtime_reader_stream.conf')
try:
    os.remove(path)
except OSError:
    pass

panel = load_appliances(config_file)

for panel_name, my_panel in panel.items():
    if(panel_name == 'reader'):
        remove_reader_module_configs()
        remove_reader_stream_configs()
        if(my_panel['enabled']):
            build_reader_module_configs(my_panel)
            build_reader_stream_configs(my_panel)
    else:
        remove_filter_module_configs(panel_name)
        remove_filter_stream_configs(panel_name)
        if(my_panel['enable']):
            build_filter_module_configs(panel_name,my_panel)
            build_filter_stream_configs(panel_name,my_panel)
            loads = my_panel['loads']
            for key, my_appliance in loads.items():
                remove_module_configs(my_appliance,panel_name)
                remove_stream_configs(my_appliance,panel_name)
                if(my_appliance['enabled']):
                    build_module_configs(my_appliance,panel_name,my_panel)
                    build_stream_configs(my_appliance,panel_name,my_panel)
                else:
                    loads = my_panel['loads']
                    for key, my_appliance in loads.items():
                        remove_module_configs(my_appliance,panel_name)
                        remove_stream_configs(my_appliance,panel_name)
C.4 Metrics Generator

Source Code C.27

Filename: metrics.py

Collaborators: Jennifer F. Switzer

Description: Single-file script of the metrics management tool.

```
#!/usr/bin/python
import sys, os
import re
import argparse
import nilmdb.client.client as nilm
import nilmdb.client.numpyclient as nilmnp
import nilmdb.client.errors as nilmError
from pymongo import MongoClient
import datetime
from bson.code import Code
import json
from dateutil.tz import tzlocal, tzutc
import numpy as np

CONFIG_PATH = "metrics_config.json"

class color:
    PURPLE = '\033[95m'
    CYAN = '\033[96m'
    DARKCYAN = '\033[36m'
    BLUE = '\033[94m'
    GREEN = '\033[92m'
    YELLOW = '\033[93m'
    RED = '\033[91m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
    END = '\033[0m'

day_delta = datetime.timedelta(days=1)
date_epoch = datetime.datetime(1970,1,1, tzinfo=tzutc())

def date_strip(date):
    return datetime.datetime(date.year, date.month, date.day)

def date_shift(date, days):
    return date_strip(date + days * day_delta)

def dates_equal(a, b):
    return a.year == b.year and a.month == b.month and a.day == b.day
```
```python
def timestamp_datetime(us):
    return datetime.datetime.fromtimestamp(us/1.0e6, tz=tzlocal())

def datetime_timestamp(date):
    return int(((date - date_epoch).total_seconds() * 1e6)

def date_interval(date):
    end = datetime.datetime(date.year, date.month, date.day, 0, 0,
    tzinfo=tzlocal())
    start = end - day_delta;
    return (start, end);

def date_check(s, pat=re.compile(r"\d{4}-\d{2}-\d{2}")):
    if not pat.match(s):
        raise argparse.ArgumentTypeError
    return s

def date_parse(s):
    parts = map(int, s.split("-"))
    return datetime.datetime(*parts)

def parse_path(filepath, check_exists=True):
    filepath = os.path.expanduser(filepath)
    if os.path.isabs(filepath):
        path = os.path.abspath(filepath)
    else:
        path = os.path.normpath(os.path.join(os.getcwd(), filepath))
    if check_exists and not os.path.exists(path):
        raise Exception("Input file/path does not exist.")
    return path

def path_check_write(path):
    path = parse_path(path, check_exists=False)
    directory = os.path.dirname(path)
    if not os.access(directory, os.W_OK):
        raise argparse.ArgumentTypeError
    return path

def path_check_read(path):
    path = parse_path(path, check_exists=True)
    if not os.access(path, os.R_OK):
        raise argparse.ArgumentTypeError
    return path

def intervals_contain(intervals, target_range):
    if len(intervals) == 0: return False  # nothing to search within
    if (target_range[1] < intervals[0][0] or target_range[0] >
        intervals[-1][1]):
        return False  # target begins before or ends after available data
    if len(intervals) == 1: return True  # target includes entire interval or
    begins/end within
```

# perform bisection search to find
mid = int(len(intervals) / 2)
if target_range[0] > intervals[mid-1][1] and target_range[1] <
   intervals[mid][0]:
   return False # target entirely between intervals
if target_range[1] < intervals[mid-1][1]: # target entirely on left of
   bisector
   return intervals_contain(intervals[:mid], target_range)
if target_range[0] > intervals[mid][0]: # target entirely on right of
   bisector
   return intervals_contain(intervals[mid:], target_range)
# target range straddles middle two intervals, try both sides
return intervals_contain(intervals[:mid], target_range) or 
   intervals_contain(intervals[mid:], target_range)

def generateMetrics(data_iter):
    data_list = list(data_iter)

    if len(data_list) == 0:
        return None

    STATE_ON = 1
    STATE_OFF = 2

    data = np.vstack(data_list)

    time = data[:,0]
    events = data[:,1]
    P_ss = data[:,2]
    Q_ss = data[:,3]

    total_ons = np.count_nonzero(events == 1) # count ON events

    total_events = 0
    duration = []
    real_power = []
    reactive_power = []

    load_on = False
    for i, event in enumerate(events):
        if i == 0 and event == STATE_OFF or i == events.shape[0]-1 and event ==
           STATE_ON:
           continue # ignore leading OFF or trailing OFF
        if event == STATE_ON: # also catches (incorrect) 2 ONs in a row
            load_on = True
            continue
        if event == STATE_OFF and not load_on: # caught (incorrect) 2 OFFs in a
           row
            continue
        # else, valid OFF event after a previous ON
        duration.append((time[i] - time[i-1])*1e-6)
real_power.append(-P_ss[i])  # use ON level change
reactive_power.append(-Q_ss[i])
total_events += 1

if total_events == 0: return None

apparent_power = (np.array(real_power)**2 + np.array(reactive_power)**2)**(0.5)
power_factor = np.divide(real_power, apparent_power)

metrics = {
    "realpower": int(np.average(real_power)),  # watts (int)
    "apparentpower": int(np.average(apparent_power)),  # watts (int)
    "powerfactor": np.average(power_factor),  # unitless (float)
    "avgduration": int(np.average(duration)),  # seconds (int)
    "totalduration": int(np.sum(duration)),  # seconds (int)
    "runs": total_events # unitless (int)
}

return metrics

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Generate and manage metric
data from load event streams.")
    parser._positionals.title = "Subcommands"
    parser._optionals.title = "Options"
    parser.add_argument('-A', '--host-nilmdb', dest="host_nilmdb",
                        default="localhost",
                        help="NilmDB Server Address")
    parser.add_argument('-a', '--host-mongo', dest="host_mongo",
                        default="localhost",
                        help="MongoDB Server Address")
    parser.add_argument('-v', '--verbose', dest="verbose",
                        action="store_true",
                        help="verbose logging of internal actions")

    subparsers = parser.add_subparsers()

    parser_list = subparsers.add_parser("list",
        description="List all loads and metrics.",
        help="list all loads")
    parser_list.set_defaults(mode="list")
    parser_list._optionals.title = "Options"
    parser_list.add_argument("-m", "--metrics", dest="list_metrics",
                            action="store_true",
                            help="list set of metrics for each load")
    parser_list.add_argument("-e", "--extents", dest="list_extents",
                            action="store_true",
                            help="show extent of time for each load (and metric)")

    parser_generate = subparsers.add_parser("generate",
Generates summary metrics from NilmDB event streams. Without arguments, calculates missing summary days until all metrics are calculated until yesterday. Rolling metrics are also generated for the last 24 hours.

Configuration file is located at %s.

`Generating daily metrics from event streams`

Parser settings:
- `--force` to re-calculate and overwrite if metric exists for day
- `--clean` to remove metrics for days not existing in event streams

Extracts daily metrics for a specified load and date range. Results are printed to STDOUT by default, and they can be redirected to a JSON file.

Destroy daily metrics for a specified load and date range.

Backup daily metrics for a specified load and date range.
description = "Backup all or part of metric database to file."
help = "backup database of daily matrics"
parser_backup.set_defaults(mode = "backup")
parser_backup.add_argument(metavar = "filepath", dest = "filepath",
  type = path_check_write,
  help = "destination file")
parser_backup.add_argument("-l", "--load", metavar = "load_id",
  dest = "load_id", type = str,
  help = "load identifier to destroy")
parser_backup.add_argument("-s", "--start", dest = "date_start",
  type = date_check,
  help = "start date, inclusive (YYYY-MM-DD)")
parser_backup.add_argument("-e", "--end", dest = "date_end",
  type = date_check,
  help = "end date, inclusive (YYYY-MM-DD)"
)

parser_restore = subparsers.add_parser("restore",
  description = "Restores database of daily metrics from file. Conflicts are
  skipped by default."
  help = "restores database of daily matrics"
) 
parser_restore.set_defaults(mode = "restore")
parser_restore.add_argument(metavar = "filepath", dest = "filepath",
  type = path_check_read,
  help = "source file")
parser_restore.add_argument("-f", "--force", dest = "overwrite",
  action = "store_true",
  help = "overwrites conflicting points"
)

# parse args from command line
if len(sys.argv) == 1:
  parser.print_help()  # show help message by default
  sys.exit(1)
.args = parser.parse_args()

if args.mode in ["extract", "destroy"] and not (args.date_start is not
  None or args.date_end is not None):
  parser.error("must specify a --start and/or --end date")

# load configurations from file
fp = open(CONFIG_PATH, 'r')
LOADS = json.load(fp)["loads"]
load2Stream = {}
stream2Load = {}

for load in LOADS:
  idn, stream = (load["load_id"], load["stream"])
  load2Stream[idn] = stream
  stream2Load[stream] = idn
# MongoDB Format:
# { Date: "Jun 1 2017",
#   Load: "CPP",
#   Metrics: {
#     Duration: 20,
#     Power: 1500
#   }
# }

if args.mode == "list":
    agger = [{"$group": {
        "_id": "$Load",
        "minDate": { "$min": "$Date" },
        "maxDate": { "$max": "$Date" }
    }}]
    mapper = Code("""
        function() {
            for (var key in this["Metrics"]){ emit(key, null); }
        }"
    )
    reducer = Code("""
        function(key, stuff) { return null; }
    ""
    )

    with MongoClient(host=args.host_mongo) as client:
        cl = client.nilm
        loads = cl.summary.distinct("Load")
        if args.list_extents:
            le = list(cl.summary.aggregate(agger))
            load_extents = {}
            for o in le:
                load_extents[o["_id"]] = o

            for load in loads:
                sys.stdout.write("%s%s%s" % (color.BOLD + color.BLUE, load, color.END))
                if args.list_extents:
                    extent = (load_extents[load]["minDate"],
                              load_extents[load]["maxDate"])
                    sys.stdout.write(" [%s -> %s]" % (extent[0].strftime("%m/%d/%Y"),
                                                      extent[1].strftime("%m/%d/%Y")))
                    count = cl.summary.count({"Load": load})
                    sys.stdout.write(" (%d Days)" % count)
                sys.stdout.write(os.linesep)
cl.results.remove({})
results = cl.summary.map_reduce(mapper, reducer, "results")
metrics = list(results.distinct("_id"))
sys.stdout.write("\t".join(metrics))
sys.stdout.write(os.linesep)

# /======= End of LIST =======/

# /================================================================================
# = GENERATE =
# ================================

if args.mode == "generate":
    nilmdb = nilm.Client("http://%s/nilmdb" % args.host_nilmdb)
nilmdbnp = nilmp.NumpyClient("http://%s/nilmdb" % args.host_nilmdb)
mongo = MongoClient(host=args.host_mongo)
cl = mongo.nilm

def remove_residual(filt):
    removed = cl.summary.remove(filt)
    if args.verbose:
        sys.stdout.write(color.RED + color.BOLD)
        sys.stdout.write("REMOVED ")
        sys.stdout.write(color.END)
        print "%d residual entries" % removed['n']

    # clean all loads not configured
if args.gen_clean:
    load_ids = map(lambda x: x['load_id'], LOADS)
    for load in cl.summary.distinct("Load"):
        if load not in load_ids:
            filt = {"Load": load}
            remove_residual(filt)

    # /*---------- Daily Metrics ----------*/
for load in LOADS:
    try:
        intervals = list(nilmdb.stream_intervals(load['stream']))
        if len(intervals) == 0:
            filt = {"Load": load['load_id']}
            remove_residual(filt)
            continue
        except: # stream doesn't exist
            if args.gen_clean:
                filt = {"Load": load['load_id']}
                remove_residual(filt)
                continue

        first_ts = intervals[0][0]
date_now = datetime.datetime.now(tzlocal())
first_date = timestamp_datetime(first_ts)
day_mark = date_strip(first_date)  # ALWAYS 00:00:00 UTC date only
day_interval = date_interval(first_date)  # at 23:59:59 of local time

# clean everything before first day
if args.gen_clean:
    filt = {
        "Load": load['load_id'],
        "Date": {"$lt": day_mark}
    }
    remove_residual(filt)

# iterate from earliest date to day before today
while day_interval[1] < date_now:
    # find metrics for target date
    filt = {
        "Load": load['load_id'],
        "Date": day_mark}  # query interval for DST
    found = cl.summary.count(filt)

    day_target_us = map(datetime_timestamp, day_interval)  # in microseconds
    exists_src = intervals_contain(intervals, day_target_us)
    exists_dst = (found > 0)

    generate, remove = False, False
    if args.gen_force and exists_src:
        remove = exists_dst
    if args.gen_clean and not exists_src:
        remove = exists_dst
    if exists_src and (not exists_dst or remove):
        generate = True

    if remove:
        cl.summary.remove(filt)
        if args.verbose:
            sys.stdout.write(color.RED + color.BOLD)
            sys.stdout.write("REMOVED %s %s " %
                (day_mark.strftime("%Y-%m-%d"), load['load_id'])))
            sys.stdout.write(color.END)
            print ""

    if generate:
        data_iter = nilmdbnp.stream_extract_numpy(load['stream'],
            start=(day_target_us[0]),
            end=(day_target_us[1]))

        metrics = generateMetrics(data_iter)

        if metrics is not None:
            doc = {
                "Load": load['load_id'],
                "Date": day_mark,
                "Metrics": metrics}
            cl.summary.insert(doc)
if args.verbose:
    sys.stdout.write(color.GREEN + color.BOLD)
    sys.stdout.write("STORED %s %s " %
                     (day_mark.strftime("%Y-%m-%d"), load['load_id'])))
    sys.stdout.write(color.END)
    print metrics

    # move to next date
    day_mark = date_shift(day_mark, 1)
    day_interval = date_interval(day_mark)

    # /*---------- Rolling Metrics (24-hour) ----------*/
    for load in LOADS:
        time_now = datetime.datetime.now(tzlocal())
        interval_rolling = map(datetime_timestamp, [time_now - day_delta, 
                                                     time_now])

        try:
            intervals = list(nilmdb.stream_intervals(load['stream'],
                                                      start=interval_rolling[0],
                                                      end=interval_rolling[1]))
        except:  # stream doesn't exist
            cl.rolling.remove({'Load': load['load_id']})
            continue

        if len(intervals) == 0:
            cl.rolling.remove({'Load': load['load_id']})
            continue

        data_iter = nilmdbnp.stream_extract_numpy(load['stream'],
                                                   start=(interval_rolling[0]),
                                                   end=(interval_rolling[1]))

        metrics = generateMetrics(data_iter)

        if metrics is not None:
            doc = {'Load': load['load_id'],
                   'Updated': datetime.datetime.now(tzlocal()),
                   'Metrics': metrics}
            filt = {'Load': load['load_id']}
            if cl.rolling.count(filt) > 1:  # impossible case (preventative)
                cl.rolling.remove({'Load': load['load_id']})
                cl.rolling.update(filt, doc, upsert=True)
            if args.verbose:
                sys.stdout.write(color.PURPLE + color.BOLD)
                sys.stdout.write("ROLLING %s %s (load['load_id'])")
                sys.stdout.write(color.END)
                print metrics
            else:
                cl.rolling.remove({'Load': load['load_id']})
nilmdb.close()
nilmdbnp.close()
mongo.close()

sys.exit(0)

# /*===== End of GENERATE ======*/
# /*===================================
# = ^ EXTRACT =
# ===============================*/

if args.mode == "extract":
    mongo = MongoClient(host=args.host_mongo)
    cl = mongo.nilm

    filt = {"Date": {}}

    if args.date_start is not None:
        day_start = date_parse(args.date_start)
        filt["Date"]["$gte"] = day_start

    if args.date_end is not None:
        day_end = date_parse(args.date_end)
        filt["Date"]["$lte"] = day_end

    if args.load_id is not None:
        filt["Load"] = args.load_id

    data_iter = cl.summary.find(filt)
    data = list(data_iter)
    for point in data:
        del point["_id"]
        point["Date"] = point["Date"].strftime("%Y-%m-%d")
    print json.dumps(data)

    mongo.close()

    sys.exit(0)

# /*===== End of EXTRACT ======*/
# /*===================================
# = ^ DESTROY =
# ===============================*/

if args.mode == "destroy":
    mongo = MongoClient(host=args.host_mongo)
    cl = mongo.nilm
filt = "Date": {}\n
if args.date_start is not None: \n    day_start = date_parse(args.date_start) \n    filt["Date"]["$gte"] = day_start \n
if args.date_end is not None: \n    day_end = date_parse(args.date_end) \n    filt["Date"]["$lte"] = day_end \n
if args.load_id is not None: \n    filt["Load"] = args.load_id \n
removed = cl.summary.remove(filt) \nif args.verbose:  
    sys.stdout.write(color.RED + color.BOLD) \n    sys.stdout.write("REMOVED ") \n    sys.stdout.write(color.END) \n    print "%d entries" % removed[\'n\'] \n
mongo.close() \nsys.exit(0) 

# /*===== End of DESTROY ======*/ 
# /*==============================*/  
# = BACKUP =  
# ==============================*/  
if args.mode == "backup":  
    mongo = MongoClient(host=args.host_mongo)  
    cl = mongo.nlml 
    filt = {} \n
if args.date_start is not None: \n    day_start = date_parse(args.date_start) \n    if "Date" not in filt: filt["Date"] = {} \n    filt["Date"]["$gte"] = day_start \n
if args.date_end is not None: \n    day_end = date_parse(args.date_end) \n    if "Date" not in filt: filt["Date"] = {} \n    filt["Date"]["$lte"] = day_end \n
if args.load_id is not None: \n    filt["Load"] = args.load_id
data_iter = cl.summary.find(filt)
data = list(data_iter)
for point in data:
    del point['_id']
    point['Date'] = point['Date'].isoformat()
with open(args.filepath, 'w') as fp:
    json.dump(data, fp)

if args.verbose:
sys.stdout.write(color.YELLOW + color.BOLD)
sys.stdout.write("BACKUP ")
sys.stdout.write(color.END)
print "%d entries" % len(data)
mongo.close()
sys.exit(0)

# /*===== End of BACKUP ======*/

# /*================================
# = RESTORE =
# ===============================*/

if args.mode == "restore":
mongo = MongoClient(host=args.host_mongo)
cl = mongo.nilm
with open(args.filepath, 'r') as fp:
    data = json.load(fp)
for doc in data:
date = date_parse(doc["Date"][10])
doc['Date'] = date
filt = {"Load": doc['Load'], "Date": date}
if args.overwrite:
    result = cl.summary.update(filt, doc, upsert=True) # inserts/overwrites
    if args.verbose:
        if result['updatedExisting']:
            sys.stdout.write(color.RED + color.BOLD + "OVERWROTE ")
        else:
            sys.stdout.write(color.GREEN + color.BOLD + "STORED ")
sys.stdout.write("%s %s%s " % (date.strftime("%Y-%m-%d"),
    doc['Load'], color.END))
    print doc['Metrics']
else:
count = cl.summary.count(filt)
if count == 0:
    cl.summary.insert(doc)
if args.verbose:
    if count > 0:
        sys.stdout.write(color.YELLOW + color.BOLD + "SKIPPED ")
    else:
        sys.stdout.write(color.GREEN + color.BOLD + "STORED ")
        sys.stdout.write("%s %s %s " % (date.strftime("%Y-%m-%d"),
→ doc["Load"], color.END))
        print doc["Metrics"]

mongo.close()
sys.exit(0)

# /***** End of RESTORE =====*/
Source Code C.28

Filename: metrics-config.json

Collaborators: Jennifer F. Switzer

Description: Example configuration file for the metrics management tool.

```json
{
  "loads": [
    {
      "name": "CPP",
      "load_id": "cpp",
      "stream": "/dashboard-streams/cpp"
    },
    {
      "name": "MDE JW",
      "load_id": "mdejw",
      "stream": "/dashboard-streams/mde_jw"
    },
    {
      "name": "MDE LO",
      "load_id": "mdelo",
      "stream": "/dashboard-streams/mde_lo"
    },
    {
      "name": "MDE PL",
      "load_id": "mdepl",
      "stream": "/dashboard-streams/mde_pl"
    },
    {
      "name": "OWS",
      "load_id": "ows",
      "stream": "/dashboard-streams/ows"
    },
    {
      "name": "SSDG JW",
      "load_id": "ssdgjw",
      "stream": "/dashboard-streams/ssdg_jw"
    },
    {
      "name": "SSDG LO",
      "load_id": "ssdglo",
      "stream": "/dashboard-streams/ssdg_lo"
    },
    {
      "name": "SSDG LO Spencer",
      "load_id": "ssdglo_spencer",
      "stream": "/spencer_stbd/ssdg_lo"
    }
  ]
}
```