Fuel-Conserving Environmental Control Strategies
for Small, Islanded Microgrids

by
Spencer C. Shabshab

Submitted to the Department of Mechanical Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Mechanical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

© Massachusetts Institute of Technology 2018. All rights reserved.

Signature redacted

Department of Mechanical Engineering
May 25, 2018

Certified by
Steven B. Leeb
Professor
Thesis Supervisor

Peter Lindahl
Postdoctoral Associate
Thesis Supervisor

Accepted by
Rohan Abeyaratne
Chairman, Department Committee on Graduate Theses
Fuel-Conserving Environmental Control Strategies for Small, Islanded Microgrids

by

Spencer C. Shabshab

Submitted to the Department of Mechanical Engineering on May 25, 2018, in partial fulfillment of the requirements for the degree of Master of Science in Mechanical Engineering

Abstract

In this thesis I designed and deployed a control system that subjects Environmental Control Units (ECUs) at U.S. Army Forward Operating Bases (FOBs) to centralized control, and demonstrated how intelligent operation of the ECU load can result in considerable fuel savings. I also developed a detailed simulation environment to simulate the operation of the FOB under a variety of operating conditions, and used it to investigate other potential solutions for the efficiency problems caused by ECUs. Finally, I formulate a concept for the autonomous and network-free coordination of ECU loads based on microgrid state inferences from local voltage measurements.

Thesis Supervisor: Steven B. Leeb
Title: Professor

Thesis Supervisor: Peter Lindahl
Title: Postdoctoral Associate
Acknowledgments

Thanks to my parents, Connie and Sam Shabshab, who made all things possible for me.
## Contents

1 Introduction
   1.1 Fuel Conservation Approaches ........................................ 21
   1.2 Research Contributions ................................................... 23

2 Microgrid Modeling and the Opportunity for Fuel Savings Through
   Centralized Load Control .................................................... 25
   2.1 System Characterization .................................................... 27
      2.1.1 Generator Characteristics ........................................... 29
      2.1.2 Tent Characteristics ................................................ 31
   2.2 Fuel Savings Opportunity .................................................. 34
      2.2.1 Simulation Testbed ..................................................... 36
      2.2.2 Simulation Parameters ................................................. 38
      2.2.3 Simulation Under Thermostatic ECU Control ...................... 39
      2.2.4 Simulation Under Centralized Control ............................ 39
   2.3 Conclusion ................................................................. 41

3 Field and Simulation-Based Explorations of Coordinated Environmental
   Control Unit Operation ....................................................... 43
   3.1 Control Hardware ........................................................... 43
      3.1.1 Controlled Device: F100-60K ECU .................................... 44
      3.1.2 Implanted Controller Overview .................................... 45
      3.1.3 Field Test Proof-of-Concept ....................................... 48
   3.2 Control Overview .......................................................... 49
3.2.1 Band-Seeking Algorithm ............................................................... 51
3.2.2 Average-Seeking Algorithm .......................................................... 55

4 Autonomous, Network-Free Coordination

4.1 Load Tracking with Voltage ................................................................. 63
4.1.1 Frequency Deviations ................................................................. 64
4.1.2 Harmonic Production ................................................................. 65
4.2 Voltage Monitoring Tests ................................................................. 66
4.2.1 Test Setup .................................................................................. 66
4.2.2 Test Procedure ........................................................................... 67
4.2.3 Voltage Analysis ......................................................................... 68
4.3 Voltage Monitoring: Event Transients ............................................... 69
4.3.1 Frequency Transients ................................................................. 69
4.3.2 Seventh Harmonic Voltage Transients ......................................... 72
4.4 Transient Identification Algorithm .................................................... 82
4.4.1 Transient Detection ..................................................................... 82
4.4.2 Transient Identification ............................................................... 83
4.4.3 Exemplar Formation .................................................................... 87
4.4.4 Identification Challenge: Collision transients ................................ 88
4.5 Control Concept .............................................................................. 89
4.5.1 ECU Behavior Under Thermostatic Control ................................... 90
4.5.2 Desired Coordination Behavior ................................................... 90
4.5.3 Simulation Environment .............................................................. 91
4.6 Autonomous ECU Control Implementation ....................................... 92
4.6.1 Action Constraints ....................................................................... 92
4.6.2 Protocols ................................................................................... 95
4.7 Testbed Overview: The Base Camp Integration Laboratory .............. 99
4.7.1 Diesel Generator Model ............................................................... 99
4.7.2 Preliminary Simulation Results ................................................... 103
5 Conclusion

A Hardware Documentation

A.1 The F100-60K Environmental Control Unit (ECU) .............................................. 107
  A.1.1 Mode Selection Circuitry ................................................................. 110
  A.1.2 Heat Mode Operation ................................................................. 112
  A.1.3 Cool Mode Operation ................................................................. 113
  A.1.4 Legacy Mode Operation Rules ....................................................... 117

A.2 Taking Control ........................................................................................................ 117
  A.2.1 Controller Installation: General ............................................................. 118
  A.2.2 ECU Controller: Revision 1 ................................................................. 120
  A.2.3 Demand Signal Relay ............................................................................. 124
  A.2.4 ECU Controller: Revision 2 ................................................................. 124
  A.2.5 Demand Signal Relay ............................................................................. 128

A.3 Temperature and Humidity Sensor ........................................................................ 128

A.4 Compressor State Monitor ................................................................................. 128

A.5 Installation Wiring Schematics .......................................................................... 129

B Software Documentation

B.1 Raspberry Pi ECU Control .................................................................................. 135
  B.1.1 ECU_Controller.py .................................................................................. 137
  B.1.2 modbus.py ............................................................................................. 141
  B.1.3 regbank.py ............................................................................................. 144
  B.1.4 Compensator.py ...................................................................................... 154
  B.1.5 Feedback.py ............................................................................................ 158
  B.1.6 ModeSwitch.py ....................................................................................... 160
  B.1.7 ADC.py ................................................................................................... 169
  B.1.8 CLK.py ..................................................................................................... 171
  B.1.9 DAC.py .................................................................................................... 177
  B.1.10 Relay.py .................................................................................................. 179
  B.1.11 TempSense.py ....................................................................................... 180
B.2 MATLAB ECU Control ........................................... 186
  B.2.1 Control_ECUs.m ........................................... 189
  B.2.2 ctrl_ECU.m ................................................. 203
  B.2.3 send_command .............................................. 209
  B.2.4 format_array.m ............................................ 214
  B.2.5 collect_data.m ............................................. 218
  B.2.6 scheduler_func.m ........................................... 221
B.3 MATLAB temperature data analysis/display ...................... 224
  B.3.1 compile_data.m ............................................ 224
  B.3.2 unify.m .................................................... 231
  B.3.3 clip_camp.m ............................................... 236
  B.3.4 controller_read.m ......................................... 238
  B.3.5 detect_cool_u.m ............................................ 241
  B.3.6 detect_cool_c.m ............................................ 243
  B.3.7 detect_heat_c.m ............................................ 245
  B.3.8 detect_heat_u.m ............................................ 247
  B.3.9 hobo_read.m ................................................ 250
  B.3.10 weather_read.m ............................................ 250
  B.3.11 analyze.m ................................................ 252
B.4 MATLAB tent thermal modeling .................................. 267
  B.4.1 grey_package.m ............................................ 267
  B.4.2 grey_concat_2.m ............................................ 281
  B.4.3 grey_fit.m .................................................. 288
  B.4.4 grey_eval.m ................................................ 295
  B.4.5 grey_master.m .............................................. 315
B.5 MATLAB FOB simulation ......................................... 321
  B.5.1 duplex.m .................................................... 322
  B.5.2 ecu.m ...................................................... 326
  B.5.3 gen.m ....................................................... 328
  B.5.4 identifier.m ................................................ 340
B.5.5 latrine_pump.m ........................................ 407
B.5.6 lc.m ..................................................... 410
B.5.7 mc.m ..................................................... 426
B.5.8 weather .................................................. 438
B.5.9 camp.m ................................................... 441
B.5.10 test_sim.m ............................................. 451

B.6 MATLAB voltage data analysis/display ....................... 455
  B.6.1 main_analysis_tabular.m ............................... 455
  B.6.2 pete_prep.m .......................................... 459
  B.6.3 Zero_Intercept_Times.m ............................... 464
  B.6.4 view_trans.m .......................................... 466
  B.6.5 anlz_3.m .............................................. 470
  B.6.6 test_identifier.m ..................................... 479
List of Figures

2-1 Aerial photograph of the BCIL [32]. ................................. 28
2-2 A typical FOB load breakdown. This load profile was captured at
the BCIL by the Deployable Nonintrusive Load Monitor (DepNILM)
during 48 hours of occupancy by 90 soldiers [15]. ...................... 29
2-3 Approximate fuel rate and efficiency curves for 1, 2, and 3 parallel-
connected 60kW diesel generators. ..................................... 31
2-4 Thermostatic ECU control regime. ................................. 32
2-5 Berthing complex model. Subscripts '1' refer to tent section 1, sub-
scripts '2' refer to tent section 2. ................................. 33
2-6 Representative example of model fit performance for a single tent section. 36
2-7 Information flow schematic of the object-oriented simulation testbed. 37
2-8 Load profile, temperature performance, and number of generators ac-
tive under thermostatic ECU control regime. ...................... 40
2-9 Block diagram of the centralized control scheme. Lines labeled with
an "8" indicate a signal bus containing a value for each of the eight
tent sections simulated. ........................................ 41
2-10 Load profile, temperature performance, and number of generators ac-
tive under synchronous ECU control regime. ...................... 42
3-1 Centralized control system diagram. ............................. 44
3-2 Local controller designed for retrofit to ECUs. ..................... 46
3-3 Weather conditions measured between May 22nd and May 23rd, 2018.
These data are used for further simulation in this study. ........... 48
3-4 Operation of four ECUs controlled under the traditional thermostatic regime. All four ECUs ran simultaneously at some points, with none running at others.

3-5 Operation of four ECUs controlled under a centralized regime. No more than 3 ECUs ran simultaneously at any point.

3-6 Operation of the band-seeking algorithm, under no peak load constraint.

3-7 Operation of the band-seeking algorithm, where the peak load constraint allows all ECUs to meet their temperature setpoint.

3-8 Operation of the band-seeking algorithm, where the peak load constraint does not allow all ECUs to meet their temperature setpoint.

3-9 Operation of the band-seeking algorithm, with heterogeneous temperature setpoints.

3-10 Block diagram of the average-seeking centralized control regime. Solid lines denote signals that update only when a new schedule is created, and dashed lines indicate signals that are updated every time step.

3-11 Thermal circuit equivalent model of a berthing complex in steady-state, used by the model-based feed forward module of the average-seeking control regime to generate a rough estimate of the duty cycle required for the next schedule period.

3-12 Operation of the average-seeking algorithm, under no peak load constraint.

3-13 Operation of the average-seeking algorithm, where the peak load constraint does not allow all ECUs to meet their temperature setpoint.

3-14 Operation of the average-seeking algorithm, with heterogeneous temperature setpoints.

4-1 Test setup for voltage measurements at the BCIL. Non-contact current sensor is labeled "A", and contact voltage sensors measuring at terminals of ECUs one and three are labeled "V".
4-2 Both "slow" and "fast" transients were observed for ECU events at some load levels.

4-3 "Fast" transients at some load levels are followed by a secondary frequency deviation of varying amplitude. At the base load captured here, the amplitude varies from approximately 0.2Hz to 0.4Hz.

4-4 The large inrush current of a latrine pump turn-on event makes its transient difficult to distinguish from that of an ECU turn-on, despite the fact that it is a less powerful load. These transients were captured with one generator running loaded between 35%-40% (21kW-24kW).

4-5 The amplitude of every recorded ECU turn-on frequency transient against the base load at which it occurred.

4-6 The amplitude of every recorded ECU turn-off frequency transient against the base load at which it occurred.

4-7 The frequency transient produced by a generator turn-on event.

4-8 The frequency transient produced by a generator turn-off event. Note that at low base loads the transient is nearly undetectable, and that the ECU turn-on frequency transient closely resembles that of a generator turn-off around a base load of 25% of the generation capacity of two generators (30kW).

4-9 Per-phase seventh harmonic circuit model of the BCIL test setup.

4-10 Simplified harmonic circuit model of the BCIL test setup.

4-11 P7 data gathered under generator operation loaded by the load bank. No ECU heaters or ventilation fans operated during these tests.

4-12 In-phase seventh harmonic voltage amplitude data gathered under generator operation loaded by the load bank and ECUs. The current injected by the ECU ventilation fans was observed to create significant additional seventh harmonic voltage in the circuit.
4-13 The latrine pump turn-on transient in P7 corresponds to its power consumption rather than to its initial inrush current. This makes it clearly distinguishable from an ECU turn-on event, which is a much larger load shift.

4-14 P7 data gathered under generator operation loaded by the load bank and ECUs. The current injected by the ECU ventilation fans was observed to create significant additional seventh harmonic voltage in the circuit.

4-15 Overview of the transient-exemplar match scoring algorithm. The observed transients in both streams (colored blue, on the left of each cell) are compared with the exemplary transients for each event of interest. Note that an event’s exemplar can contain multiple transients in one stream, as illustrated in this figure for the ECU turn-on frequency stream.

4-16 The exemplar management algorithm compares a transient labeled with event type and base load to all transients in the corresponding exemplar for each stream. The measured transient will either update an existing exemplary transient (+ sign) or be added as a new one.

4-17 Simultaneous events create transients that cannot be directly identified. Here a generator and ECU turn on at the same time, creating a transient resembles neither event’s exemplar. A more advanced recognition algorithm could potentially recognize this as a combination of the transients depicted in the frequency stream by Figures 4-2 and 4-7.

4-18 An example of coincidental compartment temperature synchronization causing large ECU peak loads. The top plot depicts how many ECU heaters are operating at every instance (solid line), and the average number of ECU heaters operating during the depicted time frame (dashed line). The bottom traces track the temperature of each compartment the ECU heaters service.

4-19 Constraints on an ECU under the autonomous control regime can be categorized as precedence, thermostatic, and availability constraints.
4-20 The collision-handling routine reverts all autonomous ECU collision participants back to their pre-collision states, restoring the accuracy of pre-collision load estimates, and determines whether or not a generator event was missed.

4-21 Summary of the DQ-frame synchronous generator model.

4-22 PI feedback control of synchronous terminal voltage and rotor speed.

4-23 Preliminary simulation results with eight ECUs operated autonomously in the presence of a latrine pump. The dotted red line indicates the average number of ECUs operating during the simulation period.

A-1 Drawing of the ECU with major components and panels indicated [5].

A-2 The contents of the F100-60K Electrical Component Access compartment.

A-3 The mode selection switch circuitry. The switch is a single-pole four-throw rotary switch labeled "S2-1" in the schematic. The common node of the switch is connected to a 24VDC control voltage bus [5]. The circuit has several points of connection to the ECU’s Terminal Block 1, which are labeled in the schematic as "TB1-x."

A-4 The F100-60K ECU heater control circuit [5].

A-5 Simplified schematic of the compressor controller from [2]. Only components which are active in control of the F100-60K ECU are shown.

A-6 Front view of the F100-60K ECU’s compressor controller.

A-7 Back view of the F100-60K ECU’s compressor controller.

A-8 Rev. 2 controller installed in the Electrical Component Access compartment of the ECU.

A-9 A close-up of the connections made by the implanted controllers to Terminal Block 1.

A-10 Top view of the Rev. 1 Controller.

A-11 Bottom view of the Rev. 1 Controller.

A-12 Schematic view of the Rev.1 ECU controller’s mode selection circuitry.
A-13 Top view of the Rev. 2 Controller. ................................. 125
A-14 Bottom view of the Rev. 2 Controller. ................................. 126
A-15 Schematic view of the Rev.2 ECU controller's mode selection circuitry. 130
A-16 Top view of the i2c-enabled temperature and humidity sensor. ...... 130
A-17 Bottom view of the i2c-enabled temperature and humidity sensor. .. 131
A-18 A simple RC filter can be used to indicate when the compressor solenoid
is and is not energized with a DC voltage level. This allows monitoring
to be accomplished with the analog-digital circuitry on the embedded
ECU controllers. ................................. 131
A-19 Wiring guide for the mode control portion of the Rev.1 controller. .. 132
A-20 Wiring guide for the mode control portion of the Rev. 2 controller. .. 133
A-21 Wiring guide for the compressor control portion of the Rev. 2 controller.134
## List of Tables

2.1 Berthing complex model parameters ........................................ 35

4.1 Match thresholds applied to all exemplars across all streams. .......... 86

4.2 DQ Machine Model Parameters ............................................... 102
Chapter 1

Introduction

The Forward Operating Bases (FOBs) of the United States Military are rugged, portable, and versatile facilities that are designed to be built-up, broken down, or moved in a matter of days. Like the servicemen and servicewomen who occupy them, FOBs must be capable of enduring and performing within exceptionally austere conditions. The U.S. Army’s Force Provider Operations manual, which exhorts the planners and operators of FOBs to consider unit preparedness for "extreme temperatures, rain, snow, wind or high elevation," is a testament to the diversity of hardships these facilities are expected to absorb [3]. Among these various challenges, the efficient use of limited and direly expensive energy resources is one of the most pressing. One gallon (3.78 L) of diesel fuel, which many FOBs rely on either primarily or solely for energy, can cost hundreds of dollars to deliver to FOBs at the tactical edge [20, 6]. Further, convoys often have to pass through hostile areas risking soldier casualties [14]. Efficient energy utilization, and a clear understanding of how decisions affect fuel consumption, are thus paramount for FOBs.

1.1 Fuel Conservation Approaches

A great deal of focus has already been brought to bear on the problem of energy conservation in FOBs. Incremental improvements to the efficiency of individual appliances such as generators, Environmental Control Units (ECUs), and thermally-
insulated tent liners, are current targets for improvement and innovation [6]. In addition to such incremental improvements, the integration of distributed renewable energy resources is also being investigated as a vector for fuel savings. One study, which included extensive and detailed simulation of a 1,100-personnel Air Force FOB, estimated that integrating an optimal amount of renewable resources into the FOB could reduce fuel consumption by 17% [25].

While the potential to offset fuel consumption with distributed renewable energy resources is considerable, increasing the penetration of renewable resources in a power system introduces its own set of challenges. Renewable resources are intermittent by nature, and if deployed replace or fill in for "dispatchable" energy resources, i.e. those that can provide power whenever required, renewable energy resources can actually exacerbate difficulties in maintaining balance between power generation and power consumption, also known as "source-load balancing." Even without a high penetration of inherently intermittent distributed renewable resources such as solar cells or wind power, small islanded microgrids are more challenged to achieve source-load balancing than large bulk power systems because their load profiles tend to be more variable as a percentage of generation capacity. Boait et al. offer a mathematical argument for this phenomenon based on the central limit theorem, and use a Monte Carlo simulation to demonstrate that the load variability of a population of similar loads increases as the population size decreases [11]. This expectation is corroborated by Sprague’s observation that the generators of islanded FOB microgrids tend to run severely underloaded because the peak demand for which they must be sized far exceeds the average power they actually supply [31].

Several research efforts geared toward fuel savings in microgrids like those of military FOBs attempt to improve generator utilization and increase fuel efficiency by reducing or accommodating the variability of the microgrid’s load profile. These approaches typically employ a combination of generator dispatch [17], energy storage [13], and load management [26]. In microgrids characterized by a high PA ratio, generators cannot dispatch quickly enough to effectively match generation to demand, and the load profile needs to be reshaped to improve overall efficiencies. This can
be accomplished either through extrinsic energy storage devices or load management. Energy storage reduces the load PA ratio by consuming and storing energy from the generators when they would otherwise be underloaded and releasing that energy to the loads when their collective demand would otherwise exceed the generation capacity. This method is effective, but in microgrids with very large PA ratios, obtaining and maintaining energy storage of sufficient capacity can be prohibitively costly [36].

In place of extrinsic energy storage devices, load management schemes often leverage the intrinsic energy storage capacity associated with loads, e.g., hot water heaters [29, 21] and heating, ventilation, and air conditioning (HVAC) systems [9]. Such load management schemes include direct load control by centralized controllers or system operators [7, 24] or demand response where load operation decisions are made locally based on market-based pricing [22]. While such schemes have been broadly investigated as part of whole building [30], individual home [35], and coordinated home energy management systems [16], comparatively little research exists applying them to FOBs. To some extent, this can be attributed to FOB structures needing to be “quick-to-erect” [6]. These structures, often tents with marginal insulation, have low thermal capacity and high thermal losses, combining to produce low thermal time constants which limit their usefulness in time-shifting loads. For example, the berthing tents of the Army’s Force Provider [3] have thermal time constants on the order of tens of minutes [37] compared to house or apartment building time constants on the order of tens or even hundreds of hours [27, 33].

1.2 Research Contributions

This thesis seeks to further the exploration of load management schemes that leverage the short thermal time constants of structures like the tents used at military FOBs. In Chapter 2 the power system of a typical military FOB, which is powered by diesel generators and has a load profile dominated by distributed environmental control units (ECUs), is characterized through field observations. Additionally, Chapter 2 contributes an actionable simulation model of the power system and uses it to identify
an opportunity for energy savings through appropriate scheduling of the ECUs. This result corroborates the observations of past FOB efficiency studies [20, 31], and adds to them a physical intuition for how unmanaged ECU loads create large demand peaks. Chapter 3 presents a control system for distributed ECUs, deployed to a prototype FOB testbed for use in the development of ECU control schemes. It also develops two alternative formulations of the distributed heater scheduling problem that are computationally simple and tolerant of model errors. Chapter 4 investigates a radically different concept for load management at FOBs, in which ECUs operate autonomously and use local voltage measurements to infer the operating state of other units. This control concept leverages techniques developed for nonintrusive load monitoring to detect and identify voltage transients corresponding to the operation of ECUs and generators [12, 28].
Chapter 2

Microgrid Modeling and the Opportunity for Fuel Savings Through Centralized Load Control

Small microgrids can derive their electrical power from a variety of energy resources. Some of these, including U.S. Military Forward Operating Bases (FOBs), use diesel generators as the primary or sole resource. In almost all cases, efficient utilization of generation resources is a high priority. This is particularly so for FOBs, for which diesel fuel resupplies come at remarkable monetary, logistical, and safety costs. Increasing the fuel efficiency of such microgrids requires not only incremental improvements to generation and load services, but also a higher-level understanding of how these components interact. This study of a typical U.S. Army FOB characterizes its power system, which is powered by diesel generators and has a load profile dominated by distributed environmental control units (ECUs). The study contributes an actionable simulation model of this power system and uses it to identify an opportunity for energy savings through appropriate scheduling of the ECUs.

Various approaches to the problem of source-load balancing in microgrids are available. Generally, they employ a combination of generator dispatch [17], energy storage [36], and load management [31]. Each balancing tactic comprises a variety of implementations: generator dispatch can be reactive to information such as microgrid
load or energy pricing, or it can be proactive based on some predictive optimization; Energy storage has a wide array of physical formats, ranging from pumped hydro storage to galvanic cells; Load management— the manipulation of loads to shape the grid demand profile— can be enacted through economic incentives, enforced simply by "shedding" noncritical loads, or executed more carefully by selecting individual loads to delay or shed based on a cost function. Each of these three balancing tactics also has limitations: generator dispatch is ideal in that it adjusts generation capacity to match load demand, but it cannot effectively respond to fast load variations; Energy storage has a comparatively fast response time, but remains costly to implement; Demand response shapes the load profile such that it can be more easily serviced by available energy resources, but must be limited so that disturbances to load service remain within acceptable boundaries. The most effective blend of these strategies depends on the realities of the microgrid in which they are deployed.

This study of a U.S. Army FOB examines generator underutilization, driven in large part by the large peak-to-average ratio of the facility’s erratic load profile, as a primary cause of fuel waste in the military microgrid. The programmatic underutilization of generation resources in isolated microgrids is well-documented [8], as is its causal connection to large demand peak-to-average ratios [10]. Many approaches to this problem seek to accommodate the erratic load profile by smoothing the load with dedicated energy storage [20], or by shedding noncritical loads according to some priority order when the need to store or supply energy exceeds the capacity of available storage resources [34]. In systems where energy storage is limited or unavailable, load shedding to smooth the demand profile ceases to be an emergency intervention and begins to become a part of normal operation. In this case, a load management strategy that takes into account the needs and flexibilities of the microgrid’s key loads, as well as the capacity and dispatch speed of available generation resources, is needed to secure adequate load service and source-load balance.

In this chapter, we characterize and model the key loads (ECUs) and generation resources of an Army FOB based on field observations. We combine these models into an FOB simulation environment to investigate excessive fuel use caused by a
base’s high peak-to-average load ratio. Unpredictable and unnecessary "stacking" (i.e. simultaneous operation) of thermostatically controlled environmental control units (ECUs), can take a generator from underloaded to heavily loaded or overloaded in seconds. The stacking does not reflect a sudden increase in the power needed for environmental control, but rather the chance overlap of individual heater "on" periods. Energy storage can help to eliminate this problem, as achieved in [20], but is not always economically or operationally feasible. We propose a load management approach that leverages the inherent storage capacity of FOB tents to impose peak load constraints and continually eliminate the load stacking behavior on the minutes timescale while still meeting comfort requirements.

Section 2.1 introduces the FOB being observed in this study, describes its operation, and develops working models of its key components. Section 2.2 identifies the costly load stacking behavior, estimates its fuel cost using the actionable simulation developed in Section 2.1, and demonstrates one example of how a centralized control scheme could eliminate unnecessary demand peaks and save fuel at little cost to thermal performance. Section 5 summarizes the observations of this study and outlines the direction of future work.

2.1 System Characterization

This study is based on observations of the Base Camp Integration Laboratory (BCIL) in Ft. Devens, MA. The BCIL is an archetypal FOB used to test new technologies for potential deployment to FOBs around the world. It contains the same generation resources, service loads, and structures typical of such facilities. The portion of the BCIL observed in this study is modeled after the Army concept of a Force Provider (FP) 150-soldier module [3]. A picture of the BCIL is included in Figure 2-1 to show the scale of the facility. The FOB energy demand is dominated by environmental control units, service loads which maintain the camp’s various tents at acceptable temperatures. As shown in Figure 2-2, during heating operation the environmental control load accounts for over 75% of the total base load [15].
A bank of parallel-connected diesel generators powers the BCIL in an islanded microgrid configuration. The generators automatically dispatch to keep the load within 30% and 80% of the online generation capacity. The purpose of this behavior is to appropriately load the active generators while also protecting them from overload in the event of sudden demand increases.

A major portion of the generator bank power output is used for environmental control of the base's various tents, most of which are berthing complexes for soldiers. A berthing complex is comprised of two adjacent sections, each of which has a thermal time constant on the order of 10 minutes and is temperature controlled individually by an ECU. During heating operation, the ECUs are on-off controlled to maintain the tent's air temperature within a predetermined band. This is a thermostatic control regime, in which changes to the heater operating state depend only on measured tent temperature and current heater state.
Figure 2-2: A typical FOB load breakdown. This load profile was captured at the BCIL by the Deployable Nonintrusive Load Monitor (DepNILM) during 48 hours of occupancy by 90 soldiers [15].

2.1.1 Generator Characteristics

Sizing and Configuration

The BCIL generators are 60kW synchronous diesel generators directly connected to the microgrid through an interruption contactor. They are connected to one another in parallel, and can turn on and off according to their automatic dispatch rules.

Automatic Dispatch Rules

At least one generator is dispatched at all times. When load demand exceeds 80% of the total dispatched generation capacity for longer than 10 seconds, an additional generator spins up and comes online. When load demand decreases below 30% of the dispatched generation capacity for longer than approximately five minutes, one generator is taken offline and spins down.
Generator Model

For the purposes of this investigation, as in [20], the diesel generators are treated as ideal electrical sources and their fuel consumption is approximated by a slope-intercept equation as in the HOMER power system simulation software [4]. The fuel consumption rate of a single generator is approximated as,

\[ \dot{m} = \dot{m}_o + \dot{m}_1 \frac{P_{load}}{P_{rated}} \]  

(2.1)

where \( \dot{m} \) is the fuel rate of the generator bank (kg/s), \( \dot{m}_o \) is the no-load fuel rate of a single generator, \( \dot{m}_1 \) is the slope at which a single generator's fuel rate increases as the normalized load moves from 0-100\%, \( P_{rated} \) is the rated power of a single generator, and \( P_{load} \) is the load supplied by the generator.

The diesel generators that power the BCIL are all of the same construction and rating, and when paralleled they share the load equally. (2.1) can therefore be expanded to describe the fuel consumption rate of multiple paralleled generators as shown in (2.2):

\[ \dot{m} = (\dot{m}_o + \dot{m}_1 \frac{P_{load}}{P_{rated}})N \]

\[ = \dot{m}_o N + \dot{m}_1 \frac{P_{load}}{P_{rated}} \]  

(2.2)

where \( N \) generators, each rated for \( P_{rated} \), are providing a \( \frac{1}{N} \) fraction of the total load \( P_{load} \) and each one consumes fuel according to (2.1). The efficiency of \( N \) generators supplying a total power of \( P_{load} \) can be computed with (2.3):

\[ \eta = \frac{P_{load1}}{\dot{m}LHV} \]  

(2.3)

where \( P_{load} \) is the total electrical power provided by the generators (kW), \( \dot{m} \) is the total fuel rate of the generators (kg/s) according to (2.2), and \( LHV \) is the lower heating value of the generator fuel (kJ/kg) [4]. Variables \( \dot{m}_o \) and \( \dot{m}_1 \) are approximated with a linear fit to typical fuel rate data points for a 60kW diesel generator [1]. The resulting
fuel rates and efficiencies of 1, 2, and 3 generators providing power along their entire operating range, as well as the values calculated from the data points of [1], are shown in Figure 2-3.

![Generator Bank Efficiency v. Load](image)

Figure 2-3: Approximate fuel rate and efficiency curves for 1, 2, and 3 parallel-connected 60kW diesel generators.

A bank of diesel generators modeled in this way and controlled according to the logic described in Section 2.1.1, gives a reasonable approximation of the diesel generators at the BCIL. In a Chapter 4, the generator model will be supplemented with a dynamic DQ-frame model.

### 2.1.2 Tent Characteristics

#### Sizing and Configuration

The BCIL infrastructure consists primarily of berthing, amenity, and storage compartments. A large portion of these compartments are berthing complexes, in which soldiers sleep. The berthing complexes, which are semi-cylindrical in shape, have a footprint of approximately 75 ft. x 25 ft. (22.86 m x 7.62 m). A complex is divided into two sections, and each section is serviced by a dedicated ECU.
ECU

An F100-60K ECU is connected to each tent section by one supply and one return air duct. The ECU can operate in either heating or cooling mode. In cooling mode, it rejects heat from the tent using a heat pump. In heating mode, the ECU operates a nominal 10kW resistive heater (field observations revealed power draws closer to 9kW), under a thermostatic regime. The thermostatic control logic and its performance are shown in Figure 2-4.

![Figure 2-4: Thermostatic ECU control regime.](image)

Thermal Characteristics

The berthing complexes are lined with an insulating layer to reduce heat loss to the environment. The two sections that make up each complex are connected by a passageway in the middle of the complex, which is a path for heat flow between tent sections.

Tent/ECU Model

A lumped parameter model of the berthing complex is depicted in Figure 2-5. Each tent section is modeled as a lumped thermal capacitance ($C_t$) connected to a thermal
reservoir representing the outdoor environment, through a constant lumped thermal impedance ($R_{te}$). The thermal capacitance of each tent section depends on its contents, which may include tables, desks, bunks, equipment, and a varying number of occupants. The thermal impedance between each tent section and the environment depends, among other things, on the quality of the seal, whether doors and windows are opened or closed, and the wind speed on the outer surface of the structure. In addition to the conductive heat loss from each tent section to the environment, there is also a path for convective heat flow between connected tent sections ($R_{tt}$). An ECU, treated as a heat source, provides the control input to the thermal system according to its thermostatic control logic. To account for the fact that hot air must propagate from the ECU output through a duct and into the tent section before any heating actually occurs, a time constant is assigned to the ECU heat input. The solar flux

![Berthing complex model](image)

Figure 2-5: Berthing complex model. Subscripts '1' refer to tent section 1, subscripts '2' refer to tent section 2.

and ECU heat inputs to each tent section are further described by (2.4)-(2.5),

$$\dot{P}_h = \alpha(P_{nom} - P_h) \quad (2.4)$$

$$P_s = kP_{sol} \quad (2.5)$$
where $\alpha$ is the time constant of the ECU heater input, $P_{nom}$ is the nominal output power of the heater, $P_{sol}$ is the solar irradiance measured at a central point of the FOB, and $k$ is a scaling constant that accounts for the location and orientation of each tent section relative to the sun. In general, $k$ has a stochastic component due to atmospheric conditions and a deterministically time-variant component due to the trajectory of the sun and layout of the facility.

Four tent complexes comprising eight tent sections were modeled according to Figure 2-5 using a nonlinear least-squares estimation algorithm which, given a known series of system inputs, searches for model parameters that minimize the error between model response and measured response. Recorded environmental temperature, solar irradiance, heater states, and tent section temperatures from a test on May 22nd, 2018 are used here to fit the thermal model of Figure 2-5 to each tent complex. The result of this algorithm for one tent section is shown in Figure 2-6. The parameter values produced for all tent complexes are shown in Table 2.1. In all cases, the average of the model response temperature was within $0.36^\circ F \ (0.2^\circ C)$ of the measured average. The model response predicted the measured maximum and minimum temperatures to within $1.26^\circ F \ (0.7^\circ C)$, and was at all times within $2.8^\circ F \ (1.56^\circ C)$ of the measured temperature. To further corroborate the fit, the model parameters fit to data from between 1600 and 2200 on May 22nd were used to predict all section temperatures from 2200 on May 22nd to 0400 on May 23rd. In this case the average temperature was estimated to within $0.77^\circ F \ (0.43^\circ C)$, the maximum and minimum to within $1.57^\circ F \ (0.875^\circ C)$, and at all times to within $2.8^\circ F \ (1.56^\circ C)$.

It is worth noting that the model also replicates an important field observation: the ECUs under thermostatic control switch on and off at slightly different frequencies and as a result drift slowly in and out of phase with one another.

### 2.2 Fuel Savings Opportunity

Because the ECUs are thermostatically controlled, and further because they do not all cycle at exactly the same frequency, they must be expected to occasionally align
<table>
<thead>
<tr>
<th>Complex:Section</th>
<th>$C_t\left(\frac{k_l}{K}\right)$</th>
<th>$R_{te}\left(\frac{K}{k_W}\right)$</th>
<th>$R_{tt}\left(\frac{K}{k_W}\right)$</th>
<th>$k_s$</th>
<th>$\alpha$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:1</td>
<td>404.42</td>
<td>1.76</td>
<td>2.88</td>
<td>14.18</td>
<td>0.018</td>
</tr>
<tr>
<td>1:2</td>
<td>387.46</td>
<td>2.27</td>
<td>2.88</td>
<td>9.82</td>
<td>0.019</td>
</tr>
<tr>
<td>2:1</td>
<td>346.44</td>
<td>2.23</td>
<td>3.96</td>
<td>12.50</td>
<td>0.023</td>
</tr>
<tr>
<td>2:2</td>
<td>367.98</td>
<td>1.81</td>
<td>3.96</td>
<td>11.70</td>
<td>0.016</td>
</tr>
<tr>
<td>3:1</td>
<td>436.90</td>
<td>2.11</td>
<td>2.37</td>
<td>19.00</td>
<td>0.011</td>
</tr>
<tr>
<td>3:2</td>
<td>332.42</td>
<td>1.90</td>
<td>2.37</td>
<td>16.10</td>
<td>0.012</td>
</tr>
<tr>
<td>4:1</td>
<td>422.13</td>
<td>1.86</td>
<td>2.95</td>
<td>22.10</td>
<td>0.011</td>
</tr>
<tr>
<td>4:2</td>
<td>444.12</td>
<td>1.92</td>
<td>2.95</td>
<td>15.80</td>
<td>0.013</td>
</tr>
</tbody>
</table>

Table 2.1: Berthing complex model parameters
in phase and demand power at the same time. This can generate demand peaks that do not correspond to the average power actually required for environmental control but rather to the chance coincidence of heater on periods. This behavior significantly increases the load profile peak-to-average ratio and can incur a considerable fuel cost. In what follows, simulation of base camp operation under both the traditional thermostatic ECU control scheme and under a centralized control scheme that prevents ECU stacking identifies the opportunity for fuel savings through central ECU scheduling.

2.2.1 Simulation Testbed

An object-oriented MATLAB simulation testbed, which includes the generator and thermal plant models described in this chapter, enables exploration of the costs and benefits associated with changing the ECU control scheme. It can simulate operation of the base camp with custom ECU control schemes under arbitrary, time-varying ambient temperature and solar irradiation conditions. Number of tent complexes, rating
of generators, and initial number of generators operating can also be customized. The simulation environment is based on the information flows depicted in Figure 2-7. The simulation testbed is a conglomerate of various modules, each of which contains its own programmable decision logic and/or dynamics. The modules are defined by and exist as properties of the FOB object, which controls the duration and timestep of the simulation. It also determines simulation parameters such as weather patterns, and the composition of the FOB in terms of tents, generators, ECUs, and other loads. The testbed allows for straightforward observation of how the various components of an FOB's power system interact, and can accommodate subsystem models of a wide range of complexities. The Generators module, for instance, could include a dynamic machine model, with arbitrary voltage a frequency regulation controls, or simply be a steady-state fuel consumption model. Any Generators module that can produce the information required by the other modules in the system can be seamlessly integrated into the simulation environment. Building the simulation testbed based on the object-oriented paradigm therefore allows for various load scenarios, load configurations, and control regimes to be rapidly evaluated.
When the testbed simultaneously models modules with very different time constants, such as a generator model that includes rotor dynamics with a timescale on the order of tenths of a second and tent thermal models with time constants on the order of ten minutes, it applies a zero-order hold to avoid wasting computational resources on simulating slow modules at unnecessarily fine timesteps. Simulating an ECU controller that operates its heater based on the temperature of a tent and the measured output frequency of a synchronous generator model, for example, requires that the generator be simulated on a timescale sufficiently small to capture its mechanical dynamics, but the tent has such a large time constant that it can be satisfactorily simulated using a much larger time step. In this case the simulation environment would hold the temperature of the tent constant while it simulates the synchronous generator until the desired time step of the tent simulation has elapsed, at which point it will simulate both the generator and the tent. This was found to increase computation speed dramatically, especially when many long-timescale systems were being simulated alongside one short-timescale module, at little cost to accuracy.

2.2.2 Simulation Parameters

The simulation in this study concerns four berthing complexes comprising eight tent sections, each section heated to a temperature setpoint of 68°F (20°C) by an ECU. The maximum aggregate load that the ECUs can produce is 72kW. A constant base load of 11kW is assumed to account for ventilation (approximately 1kW per ECU), lighting, and miscellaneous small loads. The ECUs are powered by a bank of two diesel generators, each rated for 60kW and operated according to the logic described in Section 2.1.1.

The environmental inputs, ambient temperature and solar irradiation, are taken from the measurements of a local weather station on May 22nd and May 23rd. The temperature varied between 51.1°F (10.6°C) and 52.3°F (11.3°C) during the observed period, from 1600 on May 22nd to 0400 on May 23rd. In each simulation, the initial turn-on time of each ECU is a uniformly distributed random variable between zero and 30 minutes.
2.2.3 Simulation Under Thermostatic ECU Control

The ECUs operate according to their thermostatic control law, turning on when the tent temperature reaches the lower bound of the comfort region and turning off when the tent temperature reaches the upper bound. As seen in Figure 2-8, all tent sections oscillate between the same temperature bounds, but at different frequencies due to differences in their thermal time constants. The fuel consumptions generated in five simulation runs average to 367.7 lbs. (166.8 kg) or approximately 56.8 gal. (215.3 L) of JP-8 fuel. The results of one of the five simulation runs is shown for reference in Figure 2-8. The top plot shows how many heaters are operating. The middle plot shows, at every instance, the temperature of the hottest tent section and the temperature of the coolest tent section (all tent section temperatures are within the two lines at all times). The bottom plot indicates how many generators are operating. A second generator dispatches the first time the ECU load exceeds 48kW, in accordance with the automatic dispatch rules. Intermittent demand peaks in excess of 36kW (the generator turn-off threshold), keep the second generator running almost continuously throughout the night.

2.2.4 Simulation Under Centralized Control

Alternative Control Regime

As an alternative to the thermostatic ECU control approach, which has been shown to generate large peak loads and waste fuel, consider a centralized control scheme in which heater run times are scheduled on the minutes timescale such that peak load is minimized. The central controller, a block-diagram of which is shown in Figure 2-9, uses environmental temperature ($T_e$), expected solar irradiance ($P_s$), the temperature setpoints ($T_{set}$), and its model of each tent section to determine what average powers ($P_{avg}$) are required. Temperature feedback of average tent section temperatures ($T_i$) corrects for small model errors so that all tent section average temperatures converge to their setpoints in steady-state. A scheduler allots time slots to each ECU such that the average power each delivers equals its recommended control effort.
Centralized Control Performance

The control scheme applied above is now applied in simulation to the same eight tents under the same weather conditions. Model error with a statistical variance of 10% from the actual model parameters is simulated in the feed-forward controller. Five simulations under this centralized control scheme demonstrate that fuel consumption can be considerably decreased to 307 lbs. (139 kg) or approximately 47.5 gal. (180 L) of JP-8 fuel—a fuel savings of 16.7%. The performance of the centralized heater control scheme is shown in Figure 2-10. Note, in the top plot of Figure 2-10, that the peak-to-average ratio of the ECU load profile is markedly improved. The middle plot shows that, with the exception of initial temperature overshoot due to model error and the integral temperature feedback, the temperature envelope of the tents is largely unchanged. In this simulation, the central controller allocated time slots such that two groups of four ECUs alternated being on and off. The middle plot of Figure
2.2.1 Block diagram of the centralized control scheme. Lines labeled with an "8" indicate a signal bus containing a value for each of the eight tent sections simulated.

2-10 contains more ripple because of this grouping, but it should be noted that the maximum and minimum temperatures are approximately the same. Because the time constants of the tent sections are all slightly different, conversion to a synchronous control basis means that each tent sees a slightly different maximum and minimum temperature.

It should also be noted that the case simulated here is a particularly dramatic one, where the central controller is able to align heater time slots times perfectly such that four heaters are on at almost all times. Independently controlled heaters certainly cannot be expected to achieve such fine alignment in time, and so under thermostatic control they generate larger demand peaks frequently enough to keep the second generator running almost continuously. In all cases, however, a centralized control regime can guarantee a minimal peak-to-average ratio and the traditional thermostatic regime cannot.

2.3 Conclusion

Theoretically, dedicated energy storage of sufficient storage and output capacity will always be able to absorb load fluctuations of any size and enable generators to run in a
Figure 2-10: Load profile, temperature performance, and number of generators active under synchronous ECU control regime.

more optimally-loaded state. In practice, it is sometimes more economical to prevent load fluctuations altogether through load management. In the case of ECU stacking explored in this study, the management could be as simple as controlling the phase of each ECU's heater on/off cycle such that overlap is minimized. During slow-changing environmental conditions, load management of this kind can be achieved at little or no cost to temperature performance.
Chapter 3

Field and Simulation-Based Explorations of Coordinated Environmental Control Unit Operation

This chapter presents a control system for distributed ECU{s}, deployed to a prototype FOB testbed for use in the development of ECU control schemes. It also develops two alternative formulations of the distributed heater scheduling problem that are computationally simple and tolerant of model errors. A method of providing actionable information about service cost and quality to the FOB commander, based on thermal models of the serviced compartments, is also presented.

3.1 Control Hardware

A control system consisting of a master controller (MC), a local area network (LAN), and eight local controllers (LCs) was designed, constructed, and deployed at the BCIL to provide a testbed for ECU control concepts. The MC is a laptop personal computer that receives feedback data from and sends control signals to the LCs via the LAN. The MC control logic and communication programming is implemented
in MATLAB. The LCs are Raspberry Pi-based devices that fit inside of the ECU’s electronics panel and are retrofit to its control circuitry. A diagram of the control system layout is displayed in Figure 3-1, and the LC hardware is detailed in Figure 3-2. The LC control logic and communication programming is implemented in Python. The LAN consists of four ethernet switches, one for each berthing complex.

![Diagram of the control system layout](image)

Figure 3-1: Centralized control system diagram.

### 3.1.1 Controlled Device: F100-60K ECU

The ECU has four operating modes: Off, Vent, Heat, and Cool. The operating mode is normally selected with a four-position switch on the ECU’s user control panel. The LC is capable of bypassing this switch and subjecting the ECU operating mode to central control.

While in "Heat" mode the ECU normally controls its resistive heater thermostatically based on its temperature setpoint, which the user sets via an external control panel. The heater is energized by a contactor, which itself is actuated by a low-power relay. The LC is capable of bypassing the ECU’s heater relay actuation circuits and controlling the relay directly. By virtue of controlling each heater directly, the MC can freely determine the temperature setpoint of each ECU.
While in "Cool" mode the ECU operates a heat pump, which is driven by a scroll compressor. A scroll compressor uses two interleaved scrolls to compress a fluid. When the scrolls are engaged, the compressor is fully loaded, and when they are disengaged the pressure releases and the compressor runs unloaded. This mechanism is used to continuously modulate the output power of the compressor between 10% and 100% without cycling power to the device. The compressor scrolls are driven by a 24VAC solenoid, which the ECU’s compressor controller normally pulse-width modulates with a period of 15 or 20s in order meet the temperature setpoint. If the need for cooling falls below 10% of the compressor’s capacity, then the ECU’s compressor controller turns the compressor off. The LC is capable of bypassing the compressor controller’s solenoid drive circuit and controlling actuation of the contactor that energizes the device, subject to the ECU controller’s fault protocols which override the LC. This gives it control of when the compressor consumes power, how much it consumes, and the temperature setpoint that it works to meet.

In all modes except for "Off," the ECU controls the speed of the compartment ventilation fan using a variable speed drive. In "Cool" mode, the ECU selects one of two speeds for an additional fan that passes air over the condenser coils. The LC currently does not control these.

For safety reasons, all of the LC’s control points are overridden in the event of an ECU fault state. For instance, an over-temperature condition trips a resettable breaker and removes LC control of the heater. In "Cool" mode, any fault condition causes actuation of the compressor contactor and scroll solenoid to immediately return to the ECU’s compressor controller. The ECU’s fault protection mechanisms are thereby left in place.

### 3.1.2 Implanted Controller Overview

The LC is designed for retrofit to ECUs, transmits and receives control information via Modbus/TCP, and exerts near total control of ECU operation. The highlighted components of the board in Figure 3-2 are explained in the text beneath.
Digital-Analog Converter (DAC)

Two on-board DACs are used to generate control signals that can be used to achieve more complete control of the ECU. For example, the ECU in which this controller is installed actuates its compressor contactor based on the state of a demand signal voltage. A DAC channel can replace this demand signal and subject the unit to central control.

Real Time Clock (RTC)

The RTC is used to generate accurate timestamps for collected data when there is no network clock available. One example of such a scenario is when the LC monitors normal ECU operation and the MC is not active.
Mode Control Relays (MCRs)

MCRs manipulate internal HVAC control circuitry to change the ECU’s operating mode. The F100-60K ECU contains a low-power analog interface to nearly all of its power systems. For example, the heater are activated by providing 24VDC to the coil of a small relay which in turn energizes the heater’s contactor coil. The LC controls this process, as well as others which dictate the operating mode of the ECU, using the MCRs. The LC can bypass the triac that normally controls the solenoid and use its own solid-state relay, mounted on the bottom of the board, to take control.

Thermocouples

Two K-type thermocouples allow the LC to sense the temperature of both the air being injected by the ECU into its tent section and the air returning to the ECU from the tent section. Like the ECU under thermostatic operation, the centralized controls system uses the return air temperature as feedback for both heating and cooling operation.

Analog-Digital Converter (ADC)

These channels are used to monitor the operation of the HVAC unit. Currently, they are used to detect system faults, track the operation of the HVAC when it is not under central control, and interface with sensors such as thermocouples.

Energy Monitoring Chip

The version of the board described in this paper includes an energy monitoring chip to allow for lab experimentation with energy tracking and voltage monitoring. It is not utilized in field testing, but voltage monitoring capabilities may be valuable for future control regimes.
Environmental Conditions, 1600 May 22nd to 0400 May 23rd

Figure 3-3: Weather conditions measured between May 22nd and May 23rd, 2018. These data are used for further simulation in this study.

3.1.3 Field Test Proof-of-Concept

Centralized Heating Control

A field test on May 22nd 2018 demonstrates the heat-mode functionality of the control system detailed in this section. It involves four berthing complexes comprising eight tent sections. In two of the berthing complexes comprising four tent sections, the ECUs control temperature by the traditional method, with no centralized control. The ECUs of the other two berthing complexes are controlled by the LCs and the compartment temperatures are managed by the MC. The centralized ECU control algorithm applied in this test was a preliminary attempt to achieve adequate tent section temperatures under steady environmental conditions. The test serves primarily as hardware validations and as experimental proof-of-concept to show the peak-shaving potential of centralized ECU control.

Berthing complexes 1-2 (tent sections 1-4) operate under the traditional thermostatic regime, and berthing complexes 3-4 (tent sections 5-8) operate under a centralized control scheme. The temperature performance and load profile of the thermostatically controlled tents are shown in Figure 3-4. Those of the ECUs under centralized control are shown in Figure 3-5. The weather conditions are those described by Figure 3-3.
Figure 3-4: Operation of four ECUs controlled under the traditional thermostatic regime. All four ECUs ran simultaneously at some points, with none running at others.

3.2 Control Overview

The goal of any environmental control regime involving a two-state heater is to turn the device on and off such that the temperature of the compartment stays within certain bounds. The ECUs traditionally accomplish this under a thermostatic control regime. This regime is very robust and guarantees acceptable temperature performance for each berthing compartment, but distributed units controlled in this way sometimes generate peak loads far in excess of the average power required for environmental control. As seen in Section 2.2, this behavior can carry a considerable fuel cost. The distinction of note in how the two groups operate is that the group under central control has a peak load.

Two centralized control schemes for heating are presented here as alternatives to the thermostatic regime. Each controls the states of distributed ECU heaters with the dual objectives of attaining acceptable temperature performance and limiting the peak load presented by the heaters. The first control scheme is band-seeking, i.e. it controls distributed ECUs to remain within a temperature range defined by a
Figure 3-5: Operation of four ECUs controlled under a centralized regime. No more than 3 ECUs ran simultaneously at any point.

median temperature and the amount of deviation from that value allowed. The second is average-seeking, i.e. it controls distributed ECUs to meet average temperature requirements. Both algorithms give absolute precedence to the peak load constraint, and allow temperature performance to vary as necessary to meet it. The algorithms differ primarily in how they determine device cycle times, what they guarantee in terms of peak load limiting, and in how they operate when the peak load constraint precludes achieving the temperature setpoint. The band-seeking algorithm does not require a model of the thermal system being controlled, and neither algorithm requires a computationally expensive optimization routine. The average-seeking algorithm can be paired with a temperature forecast and thermal models of the controlled compartments to predict temperature performance given a peak load constraint, or vice versa.

Both control regimes developed in this study guarantee a maximum peak load and can achieve temperature performance comparable with the thermostatic regime, given sufficient energy resources. The control principle of both is to leverage the thermal capacity of the compartments serviced by the distributed ECUs in scheduling
distributed heater "on" and "off" times for minimal overlap. A central node controls
the cycles of the distributed ECUs, scheduling each either on a continuous basis to
meet a temperature band setpoint (band-seeking) or at 15-minute intervals to match
an average temperature setpoint (average-seeking).

3.2.1 Band-Seeking Algorithm

The band-seeking centralized ECU control algorithm attempts to control distributed
ECUs such that all tent section temperatures $T$ remain within $\Delta$ degrees of the tent
section setpoint $T_s$ as in 3.1, subject to the peak load constraint of 3.2,

$$-\Delta < T - T_s < \Delta$$  \hspace{1cm} (3.1)

$$\sum_{n=1}^{N} \delta_i \leq N_{\text{max}}$$  \hspace{1cm} (3.2)

where $\delta_i$ is equal to 0 if the $i^{th}$ heater is off and equal to 1 if it is on, and $N_{\text{max}}$
is the maximum number of heaters permitted to operate simultaneously. Individual
compartments are permitted to have different temperature setpoints but $\Delta$ is equal
among the tents. This control logic is described by Algorithm 1.

Under the band-seeking regime, the MC turns off any heater when the temperature
of its compartment rises above the allowed temperature range. The MC turns on the
heater of a compartment when its temperature has fallen below range and the number
of heaters operating is not already at its limit. If the compartment temperature is
below range and the number of heaters operating is at its limit, the MC searches for
a tent with an energized heater that is nearly at the top of its temperature band.
If the MC finds such a tent, it turns that tent section's heater off and turns on the
heater of the below-range tent section.

The main benefit of the band-seeking centralized control regime is that it can
achieve tight compliance with temperature boundaries and respond instantly to sys-
tem disturbances such as bursts of sunlight or gusts of cold wind. Its chief limitation
is that it readily runs as many heaters simultaneously as the peak load constraint
Algorithm 1 Band-seeking ECU control logic.

**Input:** Temperature setpoints $T_s$, compartment temperatures $T$, temperature bandwidth $\Delta$

```
for all tents do
  i = tent #
  $e_i = T_{s,i} - T_i$ {compute temperature deficit}
end for

for all tents with heater on do
  i = tent #
  if $T_i > T_{s,i} + \Delta$ then
    turn heater $i$ off
  end if
end for

for all tents with heater off (in descending order of $e$) do
  i = tent #
  if $T_i < T_{s,i} - \Delta$ then
    if # heaters on < limit then
      turn heater $i$ on
    else
      for all tents with heater on do
        k = tent #
        if heater i off and $T_{s,i} - T_i > T_{s,k} - T_k + 1.5\Delta$ then
          turn heater $k$ off
          turn heater $i$ on
        end if
      end for
    end if
  end if
end for
```
allows, even when doing so is not necessary. The performance of the controller in various conditions is demonstrated in simulation Figures 3-6-3-9 below.

**Peak load unconstrained**

Under no peak load constraint, or when the peak load constraint is sufficient to accommodate simultaneous operation of all ECUs, the band-seeking algorithm is identical to the thermostatic regime: heaters operate until the controlled compartment temperature reaches the upper temperature bound, and then turn off until the controlled compartment temperature reaches the lower temperature bound. The results are displayed in Figure 3-6.

![Simulated ECU Load Profile with Band-Seeking ECU Control](image)

![Simulated Temperature Performance with Band-Seeking ECU Control](image)

Figure 3-6: Operation of the band-seeking algorithm, under no peak load constraint.

**Peak load constrained, ECUs can achieve setpoints**

When the peak load constraint does not allow simultaneous operation of all heaters, the MC attempts to meet all temperature setpoints by alternating the heaters. The band-seeking algorithm enforces a maximum limit on the peak load (six, in this case) but does not necessarily minimize the peak load. The results are displayed in Figure 3-7.

53
Figure 3-7: Operation of the band-seeking algorithm, where the peak load constraint allows all ECUs to meet their temperature setpoint.

**Peak load constrained, ECUs cannot achieve setpoints**

If the peak load is constrained such that the heaters cannot achieve their temperature setpoints, then the MC allows all tent sections to fall equal amounts below their temperature setpoints. This is accomplished by the control despite the slightly different thermal characteristics of the tent sections. The results are displayed in Figure 3-8.

**Heterogenous temperature setpoints**

The band-seeking algorithm is capable of operating ECUs to meet a different temperature setpoint in each tent section. The MC determines heater priority by the temperature deficit of each compartment, which is defined as the difference between that compartment's setpoint and its measured temperature. Here, each complex has one tent section maintained at a median temperature of 18°C and one maintained at a median temperature of 20°C. The results are displayed in Figure 3-9.
Figure 3-8: Operation of the band-seeking algorithm, where the peak load constraint does not allow all ECUs to meet their temperature setpoint.

### 3.2.2 Average-Seeking Algorithm

The average-seeking centralized ECU control algorithm attempts to schedule distributed ECUs such that all compartment temperatures averaged over one full schedule period, \( \bar{T} \), equal the temperature setpoints \( T_s \). This goal is described by (3.3),

\[
\bar{T} = \frac{1}{t_o - t_0} \int_{t_0}^{t_o} T \, dt = T_s
\]

(3.3)

where \( t_o \) is the start time of a schedule period and \( \tau \) is the duration of all schedule periods. Like the band-seeking algorithm, the average-seeking algorithm is subject to the peak load constraint described by 3.2.

Individual compartments are permitted to have different temperature setpoints under the average-seeking control regime. The control logic is described at a high level by Figure 3-10. Under the average-seeking control regime the ECUs operate with their heaters controlled by the LCs, which take commands from the MC. The MC sends on/off commands to the LCs each timestep based on a schedule that it renews at regular intervals. The schedule is created by the MC's scheduling algorithm, which
Figure 3-9: Operation of the band-seeking algorithm, with heterogeneous temperature setpoints.

Figure 3-10: Block diagram of the average-seeking centralized control regime. Solid lines denote signals that update only when a new schedule is created, and dashed lines indicate signals that are updated every time step.

breaks each interval into a series of segments and assigns an on-off state to each ECU for each segment. The scheduling algorithm is described by Algorithm 2.

The duty cycle limiter applies a maximum value constraint to the duty cycles that will be passed to the scheduler. The initial maximum duty cycle constraints are equal among ECUs and are determined according to 3.4,

$$D_{max} = \frac{N_{max}}{N}$$  \hspace{1cm} (3.4)$$

where $D_{max}$ is a vector whose $i^{th}$ entry represents the maximum duty cycle allowed to
Algorithm 2 ECU scheduling logic for the average-seeking control regime

**Input:** Duty cycle requests \( d \), number of time segments \( K \)

allocate segments 1 through \( d \) to ECU 1

**for** ECUs 2 through \( N \) **do**

\( x = \text{segment after last allocated} \)

**if** \( x + d_i - 1 \leq K \) **then**

allocate segments \( x \) through \( x + d_i - 1 \) to ECU \( i \)

**else**

allocate segments \( x \) through \( K \) to ECU \( i \)

allocate segments 1 through \( d_i - K + x - 1 \) to ECU \( i \)

**end if**

**end for**

ECU \( i \) (%), \( N_{\text{max}} \) is the maximum number of ECUs allowed to operate simultaneously, and \( N \) is the total number of ECUs.

As the ECUs operate, the maximum duty cycles are allowed to change every time a new schedule is created. The MC calculates the new maximum duty cycle for each ECU by first integrating the temperature error over the previous time schedule as expressed in (3.5), then by applying the adjustment described by (3.6).

\[
e_i = \int_{t-\tau}^{t} (T_i - T_{s,i}) \tag{3.5}
\]

\[
D_{\text{max},i} = D_{\text{max},i} + \alpha \cdot \left( e_i - \frac{\sum_{k=1}^{N} e_k}{N} \right) \tag{3.6}
\]

where \( \tau \) is the duration of one schedule period, \( e \) is a vector whose \( i^{th} \) entry represents the integrated temperature error of tent section \( i \) over the previous schedule period, and \( \alpha \) is a constant scalar.

This behavior redistributes the total available heater operation time among the ECUs so that, if the peak load constraint prevents all compartments from reaching their average temperature setpoints, all will fall equal amounts below their setpoints.

The integral temperature feedback module adds to the duty cycles suggested by the model-based feed forward module \( D_o \) to correct for model error. At every time step temperature error is integrated as 3.7, subject to the anti-windup constraint of 3.8. At the end of each schedule period, when the scheduler requests duty cycles to
form its next schedule, the integral temperature feedback module provides a set of duty cycles \( (D') \) according to 3.9.

\[
D_{int} = D_{int} + k_i \cdot (T - T_s) \quad (3.7)
\]

\[
D_{int,min} \leq D_{int} \leq D_{int,max} \quad (3.8)
\]

\[
D' = D_o + D_{int} \quad (3.9)
\]

The model-based feed forward module provides an estimate of what duty cycle each heater must operate with in order to maintain the desired average temperatures in each tent. This estimate is based on the steady state model of a berthing complex, described in Figure 3-11.

![Thermal circuit equivalent model of a berthing complex in steady-state](image)

**Figure 3-11:** Thermal circuit equivalent model of a berthing complex in steady-state, used by the model-based feed forward module of the average-seeking control regime to generate a rough estimate of the duty cycle required for the next schedule period.

The steady-state relationship between the thermal characteristics of the berthing complex, the heat injected into each tent section by the ECUs \((P_{h1}, P_{h2})\), environmental temperature \((T_e)\), heat to each tent section by solar irradiation \((P_{s1}, P_{s2})\), and the temperature of each tent section \((T_{t1}, T_{t2})\) is described by 3.10,

\[
\begin{bmatrix}
  r_{11} & r_{12} \\
  r_{21} & r_{22}
\end{bmatrix}
\begin{bmatrix}
  P_{h1} \\
  P_{h2}
\end{bmatrix}
+ 
\begin{bmatrix}
  P_{s1} \\
  P_{s2}
\end{bmatrix}
= 
\begin{bmatrix}
  T_{t1} \\
  T_{t2}
\end{bmatrix}
- 
T_e \begin{bmatrix}
  1 \\
  1
\end{bmatrix} \quad (3.10)
\]

where \(r_{11}, r_{12}, r_{21}, \) and \(r_{22}\) are equivalent impedances that describe the effect of heat injection by either ECU on the steady-state temperature of either tent section, and
can be derived from circuit analysis of Figure 3-11.

The feed-forward control module estimates the \((P_{h1}, P_{h2})\) required to maintain the tent sections at setpoint by attaining the unconstrained linear least-squares solution to 3.10. The problem is formulated as shown in 3.11,

\[
\begin{bmatrix}
P_{h1} \\
P_{h2}
\end{bmatrix} = \begin{bmatrix}
r_{11} & r_{12} \\
r_{21} & r_{22}
\end{bmatrix}^{-1} \begin{bmatrix}
T_{s1} \\
T_{s2}
\end{bmatrix} - T_e \begin{bmatrix}
1 \\
1
\end{bmatrix} - \begin{bmatrix}
P_{s1} \\
P_{s2}
\end{bmatrix}
\]  

where \((T_{s1}, T_{s2})\) are the temperature setpoints for the tent sections, \(T_e\) is the expected average environmental temperature for the next schedule period, and \((P_{s1}, P_{s2})\) are the expected heat inputs to the tent sections during the next schedule period.

In some cases the feed-forward module described above will produce negative values for \((P_{s1}, P_{s2})\), requiring that heat be rejected by one of the ECUs. This can occur when the temperature setpoints for adjacent tent sections are so different that they cannot be maintained in equilibrium without one of the ECUs actively cooling due to the thermal coupling of the tent sections. Because it is unlikely and impractical to hold two adjacent tent sections at dramatically different temperature setpoints, and because using a constrained least-squares solver increases computational expense, solutions of less than zero are simply set to zero. Similarly, solutions that exceed the rating of the ECU heater are set equal to the rating.

The relationship described in 3.10 is only truly valid in steady-state, which cannot in general be expected for a berthing complex. However, given a reasonable thermal tent models and weather forecasts for the next schedule period, the feed-forward control allows the MC to respond quickly to predicted temperature or irradiance changes. It also biases the control effort near the correct steady-state values and allows tighter anti-windup constraints to be imposed on the integral feedback module.

The primary benefit of using the average-seeking algorithm is that the scheduler can guarantee not only compliance with a maximum heater overlap constraint, but further that heater overlap will be minimal at all operating points. Its chief limitation is that it cannot, in its current form, react to disturbances within a scheduling period.
The transient performance of the control is therefore limited by its PWM formulation.

**Peak load unconstrained**

The average-seeking algorithm minimizes the peak load any given set of ECU power requirements. Here there is no peak load constraint imposed on the MC, but no more than 5 ECUs must operate simultaneously in order to achieve the temperature setpoint in all tent sections. This is demonstrated in Figure 3-12.

![Figure 3-12: Operation of the average-seeking algorithm, under no peak load constraint.](image)

**Peak load constrained, ECUs cannot achieve setpoints**

If the peak load is constrained such that the heaters cannot achieve their temperature setpoints, then the MC allows all tent sections to fall equal amounts below their temperature setpoints. This is shown in Figure 3-13.

**Heterogeneous temperature setpoints**

The average-seeking algorithm is capable of operating ECUs to meet a different temperature setpoint in each tent section. Here, each complex has one tent section
Figure 3-13: Operation of the average-seeking algorithm, where the peak load constraint does not allow all ECUs to meet their temperature setpoint.

maintained at an average temperature of 18°C and one maintained at an average temperature of 20°C. The control performance in this scenario is displayed in Figure 3-14.
Figure 3-14: Operation of the average-seeking algorithm, with heterogeneous temperature setpoints.
Chapter 4

Autonomous, Network-Free
Coordination
of Environmental Control

This chapter proposes a network-free autonomous control framework in which distributed on-off controlled heaters coordinate their operation with one another such that each acts to achieve a temperature setpoint in its own compartment, and the heaters collectively do not exceed a peak load constraint. Local ECU controllers make their heater on/off decisions using a load and generation capacity estimate based on local line voltage observations.

4.1 Load Tracking with Voltage

Autonomous coordination requires some amount of shared information between agents. For ECUs to coordinate such that the power requirements of each are satisfied and the collective peak load of the ECUs is minimized, shared knowledge of the available generation capacity and how many ECUs are on at every moment is particularly useful. In some microgrids it is possible to attain this information through voltage monitoring.

From a power quality perspective, the ideal grid voltage is single-frequency, simu-
soidal, and totally independent of what happens in the grid. In real power systems, however, this cannot reasonably be achieved. The inability of generation to respond instantaneously to changing power demand results in frequency deviations from the desired value, and both generators and loads create harmonic voltage oscillations at multiples of the fundamental frequency. These power quality issues are more pronounced in islanded microgrids, which tend to see greater frequency deviations in response to load changes [23], and more harmonic distortion of the grid voltage [18]. Although these issues can never be eliminated completely, they can sometimes be harnessed to track and identify major grid events. Knowledge of major grid events, in turn, can be the basis for the shared knowledge ECUs require to coordinate autonomously.

4.1.1 Frequency Deviations

Power input to the generator’s motor must be equal in steady-state to the total electrical power output of the generator, plus all of the electrical and mechanical power dissipated by the machine. When input and output powers are not balanced in this way, the generator is in a transient condition. Because energy must always be conserved, that power imbalance means that a balancing amount must be drawn from, stored in, or dissipated by something in the machine. One part of the machine which provides that balancing power is the rotor. It can accelerate to store more kinetic energy when the machine power input temporarily exceeds its output, or slow down to supply some of its kinetic energy when the machine power output temporarily exceeds its input. In synchronous machines such deviations of the rotor speed always come with deviations in the electrical output oscillation frequency because the two are directly related. Most generators utilize some form of mechanical or digital feedback to adjust mechanical power input to the generator, in order to arrest frequency deviations and restore the nominal value. Frequency deviations can neither be detected instantaneously nor be arrested perfectly, however, so sufficiently large and sudden disturbances to the power balance in the generator tend to create detectable frequency deviations. The magnitude and shape of the deviation depend on the ro-
tational inertia of the rotor, the magnitude and abruptness of the disturbances, and
the control law of the frequency regulator.

An ECU heater at the BCIL consumes approximately 16% of the rated power of
a single 60kW diesel generator, so an ECU turn-on or turn-off is expected to cause
a significant frequency signature. Frequency deviations can also occur during the
period of time when a generator turns on and begins to parallel with another. This
is sometimes referred to as a swing transient.

4.1.2 Harmonic Production

The ideal generator outputs voltage with no harmonic distortion because the time-
varying flux linkage in each phase of its stator windings is perfectly sinusoidal. Phys-
ical generator constructions cannot achieve this ideal, however, and usually allow
some degree of voltage harmonics. Some sources of harmonic distortion in generators
can change with the generator load. For example, most synchronous generators are
constructed with magnetic materials that have a nonlinear relationship between the
strength of the magnetic field produced by the rotor and the flux linkage it creates
in the stator coils. The effect of this nonlinearity is a voltage distortion that depends
on the magnetic field strength inside of the machine. Because changing the load on a
generator inevitably changes the operating point of the magnetic circuit inside of the
machine, the voltage distortion caused by magnetic nonlinearities can be expected to
vary with load. Although generators are typically designed to operate under condi-
tions in which their materials can be considered magnetically linear to engineering
precision, the nonlinearities can still have a measurable effect on the voltage. In other
words, the voltage distortion due to magnetic nonlinearities can be simultaneously
negligible from a power quality perspective and significant from a voltage monitoring
perspective.

For the purposes of this study, each harmonic voltage oscillation is broken into
two components: An in-phase component (P), whose zero crossings coincide with the
zero crossings of the fundamental voltage oscillation, and a quadrature component (Q)
which is ninety degrees out of phase with the in-phase component. If the fundamental
voltage oscillation is described by

\[ v_o(t) = V_o \sin(\omega_o t) \]  \hspace{2cm} (4.1)

then the \( n^{th} \) harmonic can be expressed as

\[ v_n(t) = a_n \sin(\omega_o n t) + b_n \cos(\omega_o n t) \]  \hspace{2cm} (4.2)

where \( a_n \) is the amplitude of the in-phase component of the \( n^{th} \) harmonic voltage and \( b_n \) is the amplitude of the quadrature component.

### 4.2 Voltage Monitoring Tests

To assess the potential of identifying microgrid events through voltage monitoring, voltage measurements were taken at the BCIL during a series of ECU and generator on and off events. Here, \( \omega_o \) is the average frequency over a small time window.

#### 4.2.1 Test Setup

The BCIL operated as an islanded microgrid during testing, powered by either one or both of two 60kW diesel generators. The ECUs were connected radially to a point of common coupling with the generators. Also included in the test setup was a bank of three-phase resistive loads, which was used to observe event signatures under a wider variety of base loads. Voltages were measured at a sample frequency of 8kHz from phase to neutral at the terminals of two ECUs by contact meters, and currents were measured at the terminals of both generators using non-contact meters. This test setup is described by Figure 4-1.

The generators are co-located and connected to one another and to the point of coupling with loads by five phase cables carrying phase, ground, and neutral connections. From the point of common coupling, bundled cables carry power to individual tent sections and their ECUs.
4.2.2 Test Procedure

The response of the line voltage to ECU and generator events was tested by repeatedly creating those events under a variety of base loads and generator configurations. The base load was controlled using the load bank, which was capable of consuming up to approximately 40kW, and the ECUs themselves, collectively capable of similar power consumption. The ECU events, of which several hundred were captured, were centrally coordinated and actuated such that each could be easily located and labeled in the gathered voltage data. ECU events were timed such that they happened consecutively at five to seven-second intervals, enough time for the frequency deviation caused by one event to settle before the next event occurred. Generator events were captured by loading the generators such that their automatic operation rules would cause a generator to dispatch on or off. Eight generator on transients were captured at base loads ranging from 51kW to 60kW (85% to 100% of one generator's rated...
power). Eight generator off events were captured at base loads ranging from 2kW to 15kW (3% to 25% of a generator’s rated power).

4.2.3 Voltage Analysis

The voltages measured at the terminals of both ECUs were analyzed for fundamental frequency and amplitude, as well as for harmonic voltage amplitudes, in windows of time that encompass six line cycles (approximately 8.3ms). The analysis window begins on the voltage sample after a zero crossing, and ends on the voltage sample before the last zero crossing of the sixth line cycle after that. The frequency during this period is estimated as

\[ f_o = \frac{N_{cycles}}{\Delta t} \]  

(4.3)

where \( N_{cycles} \) is six in this case and \( \Delta t \) is the time elapsed between the first and last samples of the analysis window. Before further analysis, the voltage waveform is re-sampled with linear interpolation to achieve an integer number of samples per line cycle. Re-sampling is conducted such that the effective sample rate is increased as little as possible and never decreased.

The re-sampled voltage data is then analyzed using the fast fourier transform (FFT) algorithm to estimate the amplitudes of the fundamental and harmonic voltage oscillations. The resampling conducted prior to FFT analysis ensures that the FFT frequency bins will coincide with the fundamental and harmonic frequencies of interest. The result of this voltage analysis algorithm is a set of time series data, each of which represents a feature of the measured voltage such as fundamental frequency or harmonic amplitude. Each time series, such as that representing the fundamental frequency, is referred to here as a "stream."
4.3 Voltage Monitoring: Event Transients

The voltage monitoring tests described in Section 4.2 confirmed the expectations of Section 4.1 by demonstrating that ECU and generator on and off events do create detectable and repeatable transients in several streams. This section describes the transients of ECU and generator on and off events in the two streams which were found to be most useful in identifying major events at the BCIL, though any stream in which events cause transients could potentially be of use. Field testing revealed that an event's transient often depends on the generator load state before it occurred, which is referred to here as the event's "base load." As such, the following event transient descriptions will include mention of how those transients change according to the base load. Figures displaying event transients will include both the event type and the base load.

4.3.1 Frequency Transients

Frequency transients were found to provide strong and unambiguous indication of all ECU off events, strong but ambiguous indication of all ECU on events, weak indication for all generator on events, and strong but ambiguous indication for some generator off events. Frequency transients therefore provide useful information, but are not adequate on their own to ensure accurate tracking of the microgrid state. In the following figures of ECU heater frequency transients, faint black lines represent individual transients measured during testing and a bold black line averages them all to represent the essential shape.

ECU Transients

When an ECU heater changes its state, a power imbalance is created which tends to alter the grid frequency as described in Section 4.1.1. ECU heaters are the largest loads at the BCIL by a large margin, so they create a larger frequency deviation than do other loads. There are only two observed exceptions to this: one is a latrine pump that is estimated to consume roughly half the power an ECU heater does, but creates
a frequency deviation of comparable size because of the inrush current it consumes when it turns on; the second is a generator off event, which at some load levels has a frequency transient strongly resembling that of an ECU on event.

When only one generator operates, ECU heater events were observed to create frequency transients that vary in amplitude from a minimum of 0.66Hz to a maximum of 1.15Hz. One interesting characteristic of ECU heater frequency transients with one generator operating is that they take on two distinct shapes: one in which the frequency deviation is arrested and brought smoothly back to its nominal value over the course of 1.5s, and one in which it is arrested aggressively and brought back to its nominal value in less than 0.5s. The "fast" transient type is followed by a secondary frequency deviation of varying amplitude. It is unclear why two different frequency transients occur, though it was observed with operation under both generators. One unknown factor that could affect the frequency transient shape is the generator frequency regulation laws, which may be nonlinear or multi-modal.

![Graph](image)

Figure 4-2: Both "slow" and "fast" transients were observed for ECU events at some load levels.

With one generator operating under a base load of 35%-40% (21kW-24kW), the typical ECU turn-on transient becomes difficult to distinguish from a pump turn-on transient. This may also be the case at other base loads. This similarity, displayed in Figure 4-4, proscribes ECU transient detection by simple deviation thresholding,
Figure 4-3: "Fast" transients at some load levels are followed by a secondary frequency deviation of varying amplitude. At the base load captured here, the amplitude varies from approximately 0.2Hz to 0.4Hz.

and encourages the integration of additional streams for event discrimination.

When both generators operate in parallel, the amplitude of an ECU turn-on or turn-off transient depends strongly on the base load. This characteristic is described by Figures 4-5 and 4-6, which plot the amplitude of every recorded ECU turn-on and turn-off transient against the base load at which it occurred. Unlike the transients when one generator was operating, these do not take on multiple shapes at any base load.

**Generator Transients**

Generator turn-on events produce a frequency transient that does not depend strongly on base load and is not replicated by any other event, but is difficult to identify accurately because it is of small amplitude. An example of a generator turn-on frequency transient is shown in Figure 4-7.

Generator turn-off events produce a frequency transient that depends strongly on the base load. At low-loads, the transient is so small that it cannot be meaningfully identified. At higher loads, generator turn-off events produce frequency transients that closely resemble an ECU turn-on event.
Figure 4-4: The large inrush current of a latrine pump turn-on event makes its transient difficult to distinguish from that of an ECU turn-on, despite the fact that it is a less powerful load. These transients were captured with one generator running loaded between 35%-40% (21kW-24kW).

In sum, generator events produce frequency transients that are either weak or ambiguous. In order to detect these events reliably, transients in additional streams should be considered.

4.3.2 Seventh Harmonic Voltage Transients

The amplitude of the seventh harmonic voltage oscillation measured at the BCIL provided strong indication of most ECU and generator events. It was found to depend on the seventh harmonic voltage source characteristic of the generators, the seventh harmonic current injected by the ECU ventilation fans, and the network impedances across which both divide. The strong dependence of the seventh harmonic voltage on the activity of power electronic circuits such as ventilation fan variable speed drives, which are relatively small compared to the ECU heaters but have an outsize influence, means that the seventh harmonic voltage is not perfectly reliable on its own as an indicator. In addition, the behavior of the seventh harmonic voltage is likely to vary from microgrid to microgrid because of differences in loads, generators, and topology. It was, however, found to be extremely useful in resolving ambiguous frequency tran-
Figure 4-5: The amplitude of every recorded ECU turn-on frequency transient against the base load at which it occurred.

Figure 4-9 was found to be a useful model for understanding how the seventh harmonic voltage behaves at the BCIL.

Because the BCIL is a microgrid with short lines on the order of 100m in length, line impedances are approximated as resistive. The $n^{th}$ ECU has heater resistance $R_{hn}$ and a ventilation fan that injects seventh harmonic current $i_{vn}$. Voltages $V_{hl}$ and $V_{h3}$ were measured at the terminals of ECUs one and three, as described in Section 4.2.1. $V_{h1}$ is included in the schematic for illustration. The $n^{th}$ ECU is connected to the point of common coupling by a line resistance $R_{ln}$. The load bank has a controllable resistance $R_{lb}$, and is connected to the point of common coupling by a line resistance $R_{lb}$. The generators have synchronous reactances $X_{g1}$ and $X_{g2}$, and are connected to one another by short phase cables (order of 10m), whose impedances are neglected. The generators are connected to the point of common coupling by line impedance $R_{lg}$. The generators act as seventh harmonic voltage sources $V_{g1}$ and $V_{g2}$. 
As described in Section 4.1.2, these may be functions of the generator load.

The ideal condition under which local harmonic voltage amplitude measurements would be most informative, all line impedances $R_{hn}$, $R_{LB}$, and $R_{tg}$ of the short connecting cables would be zero and the voltages measured at the terminals of each ECU would be identical. In reality, the ECUs will each measure slightly different voltages at their terminals and the harmonic voltage transient of an event will not be the same everywhere. However, the line impedances at the BCIL were small enough that ECU and generator turn-on and turn-off event transients were reasonably uniform across all measurement points. Therefore, to more concisely conceptualize the behavior of the seventh harmonic voltage in the microgrid, the line impedances $R_{hn}$, $R_{LB}$, and $R_{tg}$ are neglected.

One further simplification of the model in Figure 4-9 is that generators one and two are identical and share the load equally. This is a reasonable approximation in steady state operation because the generators are of the same make and are configured to share the load equally. This simplified harmonic circuit model of the BCIL is shown in Figure 4-10.

In the simplified harmonic circuit model, all ECU heater impedances and seventh harmonic current injections are lumped into $R_h$ and $i_v$, respectively. $R_h$ is determined
Figure 4-7: The frequency transient produced by a generator turn-on event.

by 4.4.

\[
R_h = \left( \frac{s_1}{R_1} + \frac{s_2}{R_2} + \frac{s_3}{R_3} + \frac{s_4}{R_4} \right)^{-1}
\]  

(4.4)

where \( s_n \) is a binary variable that is equal to one when the \( n^{th} \) ECU heater is energized and zero when it is not. When all heaters are de-energized, then, \( R_h \) is infinite and the heater branch is an open circuit. Because the individual generator source characteristics are assumed to be identical, the seventh harmonic voltage produced by either generator operating alone can be described by 4.5.

\[
V_1(L) = V_2(L) = V(L)
\]

(4.5)

where \( V_1(L) \) and \( V_2(L) \) are the seventh harmonic voltages generated by generators one and two, respectively, as a function of the load \( L \) on each. When the generators operate in parallel, they share the load equally and are assumed to exhibit the same source characteristics of 4.5. The source characteristics of both generators can therefore be combined as in 4.6.
Figure 4-8: The frequency transient produced by a generator turn-off event. Note that at low base loads the transient is nearly undetectable, and that the ECU turn-on frequency transient closely resembles that of a generator turn-off around a base load of 25% of the generation capacity of two generators (30kW).

$$V_n(L) = V\left(\frac{L}{n}\right)$$  \hspace{1cm} (4.6)

where $n$ is the number of generators operating (one or two in this test). When both generators operate in parallel ($n=2$) under a load $L$, they each produce seventh harmonic voltage according to the 4.5 but bear only half of the total load. Because the generators output approximately the same voltage, their synchronous impedances can also be lumped according to 4.7.

$$X_g = \left(\frac{g_1}{X_{g1}} + \frac{g_2}{X_{g2}}\right)^{-1}$$  \hspace{1cm} (4.7)

where $g_n$ is a binary variable that is equal to one when the $n^{th}$ generator is operating and zero when it is not. At least one generator must be operating at all times, so at least one of $g_1$ and $g_2$ will always be equal to one. The simplified harmonic circuit of
The first tests conducted at the BCIL to measure event transients in the seventh harmonic voltage involved loading the generators over portions of their operating ranges with the load bank. Of the two components of the seventh voltage harmonic described in Section 4.2.1, the amplitude of the in-phase component (referred to as P7) was found to be the most useful for transient identification purposes. Thus, it is P7 which is displayed in the following figures. To isolate the effects of the generator source characteristics and microgrid impedances, the ECUs heaters and ventilation fans were all turned off. The two ECUs at which voltage measurements were taken were left connected to the generators. The P7 streams captured at both measurement points during these tests are shown in Figure 4-11.

The data points in this plot were computed by averaging P7 over a period of approximately five seconds for each measurement point. The data was gathered on three different days (September 12th, 14th, and 26th, 2018), under three different
Figure 4-11: P7 data gathered under generator operation loaded by the load bank. No ECU heaters or ventilation fans operated during these tests.

generator configurations (generator one only, generator three only, both generators in parallel), and from two simultaneous measurement locations (ECU one terminals and ECU three terminals). Data gathered when only generator one was operating are represented with asterisks, data gathered when only generator two was operating are represented with 'x' shapes, and data gathered when both generators operated in parallel are represented with circles. The x-axis of this plot is the total load on the generators, normalized to the total capacity of the generators as configured. Thus, a data point gathered when two generators operated in parallel under a 30kW load is located on the same vertical as a data point gathered when one generator operated alone under a 15kW load. The purpose of this normalization is to show the accuracy of the lumped generator approximation of 4.6. Voltages measured at ECUs one and three were very similar, never varying by more than 0.1V. The greatest difference between the datasets displayed is between the harmonic voltages produced by generators one and three when they operated alone near the top of their capacity. There, a difference in the amplitude of the in-phase seventh harmonic voltage produced differed by a maximum of approximately 0.2V.
When the generators are loaded not only by the load bank but also by the ECUs, the seventh harmonic current injected into the circuit by each ECUs' ventilation fan also creates a voltage drop across the microgrid impedances. In the model of Figure 4-10, the seventh harmonic voltage measured at each ECU due to the ventilation fan current injection depends on the parallel impedance of all loads and generator synchronous reactances. The results of tests conducted with the generators loaded by both ECUs and the load bank on November 21st, 2018, are shown in Figure 4-12.

![In-Phase Seventh Harmonic Voltage Amplitude (P7) vs. Load](image)

Figure 4-12: In-phase seventh harmonic voltage amplitude data gathered under generator operation loaded by the load bank and ECUs. The current injected by the ECU ventilation fans was observed to create significant additional seventh harmonic voltage in the circuit.

The results displayed in Figure 4-12 are consistent with the simple circuit model of Figure 4-10. The seventh harmonic current injected by the ECU ventilation fans creates additional voltage drops in the circuit, the magnitude of which depend on the total circuit impedance. When two generators operate, the total circuit impedance is lower and as a result the voltage change due to current injection is smaller. When only one generator operates, the total circuit impedance is higher and the injected current has more of an effect on the voltage. Note also that the generator source characteristics as captured on November 21st are consistent with those observed in
the previous tests.

P7 is immensely useful in identifying event transients unambiguously. Take, for example, the events that created ambiguous frequency transients in Section 4.3.1: when P7 measurements are added, the ambiguity vanishes. As shown in Figure 4-13, the P7 transient is related to the load change caused by the event and so clearly distinguishes between an ECU turn-on and pump turn-on.

![Figure 4-13](image)

Figure 4-13: The latrine pump turn-on transient in P7 corresponds to its power consumption rather than to its initial inrush current. This makes it clearly distinguishable from an ECU turn-on event, which is a much larger load shift.

P7 measurements also create a clear distinction between ECU turn-on and generator turn-off events, and give clearer indication of generator turn-off events that are difficult to detect and identify in the frequency stream. This is demonstrated in Figure 4-14.

In the seventh harmonic stream, ECU turn-on and generator turn-off events look very different. The simplified circuit of Figure 4-10 supports this difference. When an ECU turns on from a base load of 25% (30kW), the load on the generators operating increases by 8%. This corresponds to a change in the in-phase seventh harmonic
voltage amplitude of approximately 0.3V. When a generator turns off under similar load conditions, two things happen: The per-generator load doubles to 50% and the circuit impedance across which seventh harmonic currents from the ventilation fans travel increases. Both of these phenomena tend to increase the amplitude of the in-phase seventh harmonic voltage amplitude, and together they create a much larger step than that of the ECU turn-on transient. A load step of approximately 54kW would be required to generate a similar seventh harmonic transient.

A controller that can measure the two streams described in this section, frequency and the in-phase seventh harmonic voltage amplitude, has access to information that can unambiguously identify events of interest and also discriminate between those and the operation of smaller loads such as latrine pumps. Section 4.4 presents an identification algorithm that exploits these transients to track events of interest at the BCIL, and addresses the challenge of superposed event transients.
4.4 Transient Identification Algorithm

Section 4.2.1 identified two streams—frequency and P7—which together provide detectable, repeatable, and distinct transients that indicate four important microgrid events: ECU heater on, ECU heater off, Generator on, and Generator off. The streams also allow other kinds of events, such as latrine pump turn-ons, to be identified and distinguished from events of interest by their transients.

A collection of the transients a particular event has been observed to cause in each stream at a certain load level is referred to as that event’s "exemplar." If an observed transient matches the exemplar of a particular event across all streams, then that event is deemed to have occurred. With a method of detecting transients and matching them to exemplars, a controller can track events happening all over the microgrid by monitoring the line voltage locally.

In this application, transient detection and identification are conducted on measured streams in analysis segments of approximately 12 seconds in duration. Transient detection is accomplished by monitoring for frequency deviations from the baseline and for edges in P7. Transients are compared to exemplars in each stream by aligning them using a cross-correlation method and then computing the squared error between them. Exemplars are formed in a similar manner using labeled event transients.

4.4.1 Transient Detection

In the frequency stream the identifier classifies a deviation from baseline by more than 0.2Hz for more than 4ms, or by more than 0.01Hz for more than 2s, as a potential transient. These criteria were chosen based on the observed noise variance and the magnitudes of the frequency transients caused by events of interest. The baseline frequency is defined when the identifier begins operation and subsequently each time a transient is detected according to Algorithm 3:

In the P7 stream, an edge corresponding to a step of greater than 0.1V is considered a transient. To detect an edge, the transient detection algorithm smoothes the data in the analysis window using a Gaussian low-pass filter of the form described by
**Algorithm 3** Frequency transient detection  
**Input:** Frequency stream $f$, number of bins $n$  
**Output:** Baseline frequency, $f_{base}$

1. generate vector $b$ of $n$ equally-spaced values between min($f$) and max($f$)
2. generate vector $c$ of zeros, equal in size to $b$
3. $c_n = \#$ samples of $f$ between $b_n$ and $b_{n+1}$
4. $i = \text{argmax}(c)$
5. $f_{base} = b_i$

(4.8) and evaluates the first difference of the result as described by (4.9)

$$s = P7 \times \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(t-\mu)^2}{2\sigma^2}}$$  
$$d_n = \frac{s_n - s_{n-1}}{\Delta t}$$

where $\sigma$, the standard deviation of the normal distribution in time, is 0.3s, corresponding to 36 samples in this application. The distribution is truncated at a duration of 3s, corresponding to 360 samples in this application.

A local maximum in the first difference of the smoothed sequence that exceeds a threshold is taken to correspond to a potential edge. In this application, the threshold is three times the standard deviation of $d$ in (4.9). In order to be considered a transient, an edge that exceeds this threshold must also be associated with a step of magnitude greater than 0.1V. The identification algorithm assesses the step size by averaging the values of $P7$ for 1.5s before and after the detected edge. The identification algorithm analyzes the first rather than the strongest qualifying edge in the analysis window, so that transients are identified as they occur.

### 4.4.2 Transient Identification

Once a transient has been detected and positioned within the analysis window, an identification algorithm compares the measured transients to exemplars for every event of interest. If the measured transient matches the exemplar for an event of interest, then the transient is identified. If not, then it is considered spurious. In this application the squared error between a measured and an exemplary transient,
normalized to the energy in the exemplary transient, is the metric for their similarity. Cross-correlation offers a generic way to determine the alignment of two signals that produces the best match. The cross-correlation between a measured transient \( x \), and a shifted or "lagged" exemplary transient \( y_m \) can be defined as in (4.10).

\[
R_{xy}(m) = \sum_{n=-\infty}^{\infty} x_n y_{n+m}
\]

The argument \( m_o \) which maximizes \( R_{xy}(m) \) indicates the lag at which the correlation between \( x \) and \( y \) is greatest, or equivalently at which the squared error between the two signals is minimized. This can be understood by considering the explicit calculation of the squared error between \( x \) and \( y_m \) shown in (4.11):

\[
E_{xy}(m) = \sum_{n=-\infty}^{\infty} (x_n - y_{n+m})^2
= \sum_{n=-\infty}^{\infty} (x_n^2 + y_{n+m}^2 - 2x_n y_{n+m})
= R_{xx}(0) + R_{yy}(0) - 2R_{xy}(m)
\]

where \( R_{xx}(0) \) and \( R_{yy}(0) \) are the zero-lag cross-correlations of \( x \) and \( y \) with un-shifted copies of themselves, known as the zero-lag autocorrelation, and \( 2R_{xy}(m) \) is two times the cross-correlation between \( x \) and \( y_m \) as defined by (4.10). \( E_{xy}(m) \), as a sum of squared values, must be greater than or equal to zero. The same reasoning can be applied to \( R_{xx}(0) \) and \( R_{yy}(0) \), which are the energies of \( x \) and \( y \) respectively. With this in mind, it is apparent that the lag \( m_o \) which corresponds to the maximum value of \( R_{xy}(m) \) is also the lag at which the squared error between \( x \) and \( y_m \) is minimized.

The algorithm presented here uses cross-correlation to find \( m_o \) and then evaluates the match between measured and exemplary transients as described in (4.12)

\[
S = \frac{\sum_{n=1}^{N} (x_n - y_{n+m_o})^2}{\sum_{n=1}^{N} (x_n)^2}
\]

where score \( S \) is the squared error between the measured event’s transient and that of
the exemplar with best alignment, normalized to the energy of the exemplar. In the case where the transient and exemplar are identical, the score would be zero. Thus, a lower value of \( S \) indicates a better match between the signals. When a measured transient is compared with an event exemplar, a score is generated in this manner for each stream in which the exemplar contains transients. This concept is illustrated in Figure 4-15.

![Figure 4-15](image)

Figure 4-15: Overview of the transient-exemplar match scoring algorithm. The observed transients in both streams (colored blue, on the left of each cell) are compared with the exemplary transients for each event of interest. Note that an event’s exemplar can contain multiple transients in one stream, as illustrated in this figure for the ECU turn-on frequency stream.

In some cases, an exemplar contains multiple transients in a single stream. This is illustrated in Figure 4-15, where the ECU turn-on exemplar has two transients in the frequency stream. In these instances, the match score of the measured transient is calculated for each exemplary transient and the best score is taken.

A measured event is identified as an event of interest when its match score with that event’s exemplar falls below certain thresholds across all streams. Match thresholds in this application are statically defined for each stream of each event type. They are lower (more stringent) in streams in which an event’s exemplar contains strong
transients. Although optimal selection of the match thresholds is a topic of ongoing study, good performance was achieved with the values in Table 4.1. The Generator turn-on frequency exemplar has a higher (more tolerant) match threshold because its signal in that stream tends to be weaker and less consistent.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq.</td>
<td>0.2</td>
<td>0.2</td>
<td>0.45</td>
<td>0.2</td>
</tr>
<tr>
<td>P7</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 4.1: Match thresholds applied to all exemplars across all streams.

If an event produces no repeatable and distinct transients in any stream, then it is not detectable. The success of this method depends on having a collection of streams for which this is never the case. Occasionally, however, an event produces repeatable and distinct transients in one stream but not in the other. An example of this is a generator turn-off event at low load, which was seen in Section 4.3.1 to have a nearly undetectable transient in frequency but a very strong transient in P7 at certain base loads. The absence of a transient, although it cannot be measured using the comparison method of (4.11), is still an indication of how well a measured event matches the exemplar in this case. A measured event whose seventh harmonic amplitude transient matches that of the exemplar for a generator turn-off, but which also contains a large frequency transient that generator turn-off events are known not to create, for example, should not necessarily be classified as a generator turn-off. Accordingly, when a measured transient is being compared to an exemplar stream that does not contain a transient, the following logic is applied: If the measured event also does not contain a transient for the stream in question, then the score for that stream is set to zero, and whether or not the measured event matches the exemplar will depend on the match of the remaining streams; If the measured event
does contain a transient for the stream in question, then the score for that stream is determined according to (4.13)

\[ S = \frac{\sum_{n=a}^{b} x_n^2}{E_o} \]  

(4.13)

where \( a \) and \( b \) are the start and end indices of the transient, and \( E_o \) is set based on the most energetic frequency deviation to be tolerated.

### 4.4.3 Exemplar Formation

In order to identify events of interest by their transients in any set of streams, the transient identification algorithm requires exemplars associated with those events. This is accomplished by an algorithm that builds a library of exemplars from a set of labeled transients. The algorithm is described schematically by Figure 4-16.

![Figure 4-16: The exemplar management algorithm compares a transient labeled with event type and base load to all transients in the corresponding exemplar for each stream. The measured transient will either update an existing exemplary transient (+ sign) or be added as a new one.](image)

The algorithm receives a transient labeled with the event that caused it and the base load at which it occurred. For each stream, it assesses whether the transient
matches any of those already stored in its exemplar for that event type and base load as described in Section 4.4.2. If the measured transient does not match any of the exemplary transients, then it is added as a new exemplary transient. If it does match one of the exemplary transients, then it is aligned with that transient using (4.10) and combined with it as described by (4.14) to update the exemplar.

\[ f_e = 0.8f_e + 0.2f_m \]  \hspace{1cm} (4.14)

where \( f_e \) is the exemplary transient and \( f_m \) is the measured transient that was determined to be a match. In this way, the exemplar library is built and allowed to change either gradually through the morphing of existing exemplary transients or quickly through the creation of new ones. The further development of this algorithm, to include the elimination of obsolete exemplary transients, and the potential integration of a formal machine learning framework, will be the subject of future work.

4.4.4 Identification Challenge: Collision transients

The primary shortcoming of the identifier algorithm is its inability to parse and identify transients caused by simultaneous major events. Such transients are termed "collision transients." One example of a collision transient is shown in Figure 4-17, in which an ECU turn-on and a generator turn-on event coincided.

Collision transients can be caused by the actions of any device, including controlled devices such as ECUs and uncontrolled devices such as generators or latrine pumps. They are not repeatable or distinct because the exact timing of the events that cause them have a large effect on the shape and extent of the transient. Collision transients are therefore observed by an identifier as large and unidentified transients. In order for the autonomous ECU control regime to be feasible, there must be a way of determining what events of interest were contained in a collision even though the transient itself was not directly identifiable. Section 4.6.2 proposes a set of operating rules that address the challenge of event collisions.
Figure 4-17: Simultaneous events create transients that cannot be directly identified. Here a generator and ECU turn on at the same time, creating a transient resembles neither event's exemplar. A more advanced recognition algorithm could potentially recognize this as a combination of the transients depicted in the frequency stream by Figures 4-2 and 4-7.

4.5 Control Concept

The event identification algorithm described in Section 4.4 enables an ECU connected to the microgrid to recognize when a generator or other ECU turns on or off. This shared knowledge, combined with a set of shared operating rules, enables autonomous coordination of the ECUs. This section describes the problems of how the ECUs are operated currently, the desired behavior of ECUs under the autonomous regime, and the environment for which this particular approach is formulated. Section 4.6 describes all of the operating rules in detail.
4.5.1 ECU Behavior Under Thermostatic Control

Under the traditional thermostatic control regime, an ECU heater turns on when the temperature of its compartment falls below one threshold, and turns off when the temperature rises above a higher threshold. This type of control creates large demand peaks that far exceed the average power required to service all compartments. Take for example a period of ECU operation recorded at the BCIL and displayed in Figure 4-18, in which multiple compartments reached their bottom temperature threshold and multiple ECUs turned on in response around the same time.

![ECU Load Profile with Thermostatic ECU Control](image)

Figure 4-18: An example of coincidental compartment temperature synchronization causing large ECU peak loads. The top plot depicts how many ECU heaters are operating at every instance (solid line), and the average number of ECU heaters operating during the depicted time frame (dashed line). The bottom traces track the temperature of each compartment the ECU heaters service.

4.5.2 Desired Coordination Behavior

Autonomous ECU control should maintain the temperature responsiveness of the traditional thermostatic regime, but improve the collective behavior of the units. Supporting diversity in compartment temperatures, such that ECUs do not all require power simultaneously, can limit the inefficient and potentially damaging demand peaks described in Section 4.5.1. It is desirable that intelligent, autonomous ECUs work individually to maintain their compartment temperatures and cooperatively to
limit their aggregate peak load.

The main objective of the control, therefore, is for the ECUs to distribute their heater operation over time so that the number of heaters operating simultaneously is limited. A protocol in which the compartments most in need of heat have their ECU heaters operate first is needed to support equitable power apportionment. The autonomous ECUs' actions must also compensate the vulnerability of each unit's transient identifier to event collisions, which was discussed in Section 4.4.4.

4.5.3 Simulation Environment

As an exploration into the kinds of autonomous coordination possible with the monitoring capabilities described in Section 4.4, a set of operating rules are devised here for an islanded microgrid similar to that of the BCIL’s South Camp (BCIL-SC).

The BCIL-SC contains eight ECUs that operate 10kW resistive heaters, each servicing a berthing tent. Its structures also include a latrine and a large classroom serviced by two ECUs of the same kind. The berthing tent ECUs are the BCIL-SC's major loads, capable of generating peak loads of 80kW. It these ECUs that will operate under autonomous control. The classroom and its ECUs are neglected presently, to be incorporated in a future study developing the autonomous control regime for ECUs servicing compartments of broadly-ranging thermal time constants. The latrine and its major load– a pump of approximately 4kW capacity– are considered in the following as an uncontrolled load.

In this application, it is assumed that the small islanded microgrid is powered by two generators similar to those used at the BCIL-SC. They are 60kW synchronous diesel generators, which dispatch on and off automatically to maintain the total load within 30% and 80% of the total generation capacity.

The automatic dispatch rules of the generators inform the control objective of the autonomous ECUs. If the thermal requirements of the berthing tents can be serviced with an average power consumption of equal to or less than 40kW, then the ECUs should autonomously coordinate such that no more than four heaters operate simultaneously. This will prevent the second generator from dispatching and maintain
a higher level of efficiency. If more power is required to meet the thermal requirements of the tents, then the ECU$s$ should be allowed to violate that peak load constraint and operate on the power of both generators. Keeping the generators within their allowed operating regions must take precedence over all. Accordingly, no ECU action that causes the generator capacity to be exceeded will be taken. As an additional safeguard against overloading, the autonomous ECUs could operate with a frequency-sensitive override that causes them to self-shed if the line frequency violates certain parameters as in [19].

Finally, the ECUs should act to avoid event collisions that create unidentifiable transients and, when collisions do occur, be able to restore their operating states to the last one known.

4.6 Autonomous ECU Control Implementation

In addition to common knowledge of how many ECUs are operating and how many generators are on, ECU controllers require a set of operating rules that cause them to coordinate in the manner described by Section 4.5. In this application, the rules for autonomous ECU operation include constraints that restrict when an ECU may initiate an on/off action, and protocols that constrain how action will be taken. The protocols also include routines that make the autonomous ECUs tolerant of the event collisions described in Section 4.4.4.

4.6.1 Action Constraints

In heat mode, the ECUs have only one degree of freedom: the state of their heaters. Under traditional control, the only constraint applied to this degree of freedom is thermostatic: the ECU can only change its state when the thermal needs of the compartment rise above or fall below a certain point. The autonomous control regime adds two additional constraints to heater operation: precedence and availability. The total of all constraints—precedence, availability, and thermostatic—are summarized in Figure 4-19.
Figure 4-19: Constraints on an ECU under the autonomous control regime can be categorized as precedence, thermostatic, and availability constraints.

**Precedence Constraint**

The precedence constraint means that no ECU may act while a routine such as a collision handling is being executed, or when a transient is being identified. An additional precedence constraint is that no ECU may operate its heater when doing so would exceed the current generation capacity ($N_{max}$ heaters is the maximum number allowed with current load capacity). This is the supreme constraint, which is reflected in Figure 4-19: no action can be initiated if a transient identification or protocol execution is already in progress, or if doing so would cause an overload.

**Availability Constraint**

The availability constraint reflects a desired limit on the ECU peak load, below that mandated by the generator capacity. In this application, it is based on the load level at which a second generator dispatches (48kW). Because each ECU has a power consumption of 10kW when its heater operates, only four ECU heaters can operate simultaneously under the availability constraint ($N_{opt} = 4$). As indicated in Figure
4-19, once a cold-start timer has expired the thermostatic constraint is capable of overriding the availability constraint if the temperature falls sufficiently short of its setpoint.

**Thermostatic Constraint**

The thermostatic constraint has similar meaning under autonomous control as under the traditional control regime. Whereas there are only two states with respect to this constraint under traditional control, however, here there are three. The states are defined in terms of the temperature error $T_e$ of the compartment an ECU serves, which is described by (4.15).

$$T_e = T_s - T$$  \hspace{1cm} (4.15)

where $T_s$ is the temperature setpoint of the compartment and $T$ is the actual compartment temperature.

The first state is defined as the region in which the magnitude of temperature error $T_e$ is less than some positive lower tolerance value $T_{db}$. This is summarized in (4.16)

$$|T_e| \leq T_{db}$$  \hspace{1cm} (4.16)

Under the traditional thermostatic control the ECU takes no action while it is in this state. Under the autonomous regime the same is almost always true, the only exception being when the precedence constraint has the ECU execute some routine.

The second state refers to an intermediate temperature band in which the magnitude of $T_e$ exceeds its lower tolerance value but is less than some positive upper tolerance value $T_m$. This is summarized in (4.17).

$$T_{db} < |T_e| < T_m$$  \hspace{1cm} (4.17)

For traditional thermostatic control, there is no concept of an upper tolerance
value and an ECU always acts to restore its compartment temperature to setpoint when it first enters this region. Under the autonomous regime, an ECU in this state may act to restore compartment temperature but is subject to the availability constraint. It will not take the desired action if doing so will bring the number of heaters operating simultaneously above $N_{opt}$. The third state refers to a temperature band in which the magnitude of $T_e$ has exceeded its upper tolerance value $T_m$. This is summarized in (4.18).

$$|T_e| \geq T_m$$

(4.18)

Under the autonomous regime, an ECU in this state will initiate an action, even overriding the availability constraint if necessary. However, it remains subject to the precedence constraints.

### 4.6.2 Protocols

When an ECU is allowed by the constraints described in Section 4.6.1 to take an action, it must do so after a delay specified by its action protocols. An ECU under the autonomous regime is also required at times to execute a scripted routine in response to outside circumstances, such as the beginning of a transient due to another event. In such cases, the precedence constraint prevents the ECU from taking any other action until the routine is completed. The autonomous ECU’s protocols can therefore be categorized as delay rules and routines.

**Delay Rules**

When the autonomous ECU initiates an action, which in this case is to turn its heater on or off, it starts a delay timer. At the expiration of the timer, it takes the planned action. When an interruption routine begins, the ECU pauses its action timer until the routine is complete. The purpose of the delay timer is to allow the ECU most in need to take action first. In this application, the ECU’s action delay $\tau_d$ depends on the temperature error of its compartment and a delay rule. The ECU action delay
rule operates on the temperature error to produce a delay time. They delay time is linearly related to the temperature error, with saturation at a minimum value. This relation is described by (4.19)

\[
\tau_d = \begin{cases} 
\tau_o \pm mT_e + \eta, & \text{if } e < \frac{\tau_o - \tau_{\text{min}}}{\pm m} \\
\tau_{\text{min}} + \eta, & \text{otherwise}
\end{cases}
\]  

(4.19)

where \(\tau_d\) is the delay time, \(\tau_o\) is a delay offset, \(m\) is the delay slope, \(\tau_{\text{min}}\) is the minimum delay at which the function saturates, and \(\eta\) is a random variable of zero mean and normal distribution used to decrease the likelihood of ECUs simultaneously taking action and colliding under cold-start conditions (when the ECUs first turn on). \(\eta\) is sampled each time a delay is computed. The \(\pm\) takes the minus if the heater is currently off, such that. If (4.19) ever yields negative \(\tau_d\), then \(\tau_d\) is recalculated as a random variable of uniform distribution on the range \([0, \tau_{\text{min}}]\).

In addition to the delay rules that determine how long an ECU must wait to turn on or off when its thermostatic and availability constraints are satisfied, there is a delay rule which limits the amount of continuous heater operation time allowed to each ECU. The purpose of this rule is to enforce resource sharing among the autonomous ECUs when they are operating under the availability constraint. The maximum on-time allowed to an ECU is determined each time it begins to operate its heater, as described by (4.20)

\[
\tau_{\text{max}} = \tau_{\text{max},o} + \alpha \bar{T}_e
\]

(4.20)

where \(\tau_{\text{max},o}\) is an offset to the maximum allowed heater operation time, \(\bar{T}_e\) is the temperature error described by (4.15) averaged over a trailing window, and \(\alpha\) is a constant term. In this application, the averaging window duration was ten minutes. Allowing the maximum allowed heater operation time to vary with temperature error promotes increased resource allocation to the tents that most require heat.
Routines

A routine is a series of scripted actions performed by the autonomous ECU. When a routine is in execution, the ECU cycles between the "Monitor/Update" and "Routine In Progress" blocks of Figure 4-19. The central routine of autonomous ECU control is execution of the transient identification algorithm described in Section 4.4, referred to as the "ID routine." The ID routine is straightforward: collect data until the transient is identified, then either continue normal operation or initiate a new routine based on the result.

If the ID routine results in identification of an event of interest (ECU or generator on/off), then the ECU resumes operation according to the thermostatic and availability constraints as outlined in Figure 4-19. Under some circumstances, the ID routine may return without identifying an event of interest. There are two possible causes for this. The first is that no event of interest occurred, and the ID routine was triggered by an unmonitored event such as latrine pump operation. The second is that one or multiple events of interest collided with one another or with an unmonitored event, resulting in an unidentifiable collision transient. The transient itself does not indicate whether there was a false positive or a collision, but the latter introduces error to the load and generator state estimates used by the autonomous ECUs to make their heater operation decisions. This is clearly unacceptable, so in the case of a negative ID result the autonomous ECUs ensure that no events of interest were missed with a "collision-handling" routine.

The function of the collision-handling routine is to reverse all controllable actions which contributed to the collision, and to determine whether any uncontrolled events of interest (generator turn-on or turn-off events, in this case) also contributed. When an unidentified transient occurs, the autonomous ECUs can be separated into two groups: those that took an action which contributed to the transient, and those that did not. Members of the former group are referred to as "collision participants," and members of the latter as "collision observers." The collision-handling routine is outlined for both groups in Figure 4-20.
Figure 4-20: The collision-handling routine reverts all autonomous ECU collision participants back to their pre-collision states, restoring the accuracy of pre-collision load estimates, and determines whether or not a generator event was missed.

The routine begins when each autonomous ECU’s ID routine returns a negative identification. Collision participants know immediately at this stage that there was a collision because the transients they measure directly following their actions do not match the corresponding exemplars. Collision observers are not yet certain whether there was a collision or just a false event detection— it only observes that an unidentified transient has occurred. In this stage, both collision participants and observers log the average value of their P7 streams for the one second preceding the transient.

In the next stage of the collision-handling routine, all collision participants reverse their actions. The function of this stage is two-fold: it reverses any ECU actions that went unidentified in the collision, thereby restoring the accuracy of each autonomous ECU’s load estimate; and it confirms to all collision observers, by the transients created by the reversals, that a collision has occurred.

In the second stage of the collision-handling routine, the autonomous ECUs assess whether or not a generator event was part of the unidentified transient’s cause. This is accomplished by comparing the steady-state values of P7 measured before the collision and after the reversal. With no difference in the number of ECU heaters
operating during these two measurements, only a few potential influences on P7 remain: uncontrolled loads such as the latrine pump, ECU ventilation fan setpoint changes, and generator events. Of these three, as described in 4.3, the effect of generator events on P7 was found to be the largest by a significant margin. An extremely improbable confluence of small loads and ventilation fan actions would be required to generate a change in P7 from pre-collision to post-reversal of magnitude similar to that created by a generator transient.

4.7 Testbed Overview: The Base Camp Integration Laboratory

To enable further development and evaluation of the transient identification algorithm described in Section 4.4, as well as of the autonomous ECU control regime presented in Sections 4.5 and 4.6, a simulation mockup of the small FOB microgrid described in 4.5.3 is developed here. The mock-up is a minimal model of the BCIL-SC, including: thermal models of its eight tent sections, each of which is serviced by an ECU; the ECUs, which are programmable with either their traditional thermostatic control logic or the autonomous control logic developed in Section 4.6; a simple latrine pump model, which simulates the effect of inrush current on the generator by consuming twice its steady-state power for 0.1s when it first starts up; a full implementation of the exemplar formation and transient identification algorithm; a simplified dynamic model of the generator bank, which produces frequency and P7 data for analysis by a transient identifier; and the ability to synthesize arbitrary patterns of environmental temperature and solar irradiation, which affect the operation of the ECUs by disturbing berthing tent temperatures.

4.7.1 Diesel Generator Model

To enable simulation testing of the autonomous control system, a basic synchronous generator model that emulates the response of the diesel generators at the BCIL to
major power events is required. This model does not attempt to precisely replicate the electromechanical dynamics of the generators, nor to mimic their particular frequency and voltage regulation control laws. Rather, it seeks only to reasonably recreate the frequency deviations expected of a small synchronous machine supplying an islanded microgrid. Without including any harmonic model elements, the simulated generator can be programmed to emulate the harmonic source characteristics of the generators at the BCIL. It is a round-rotor, two-pole, dq-frame machine model. The governing equations for a single machine connected to a resistive load are expressed in Figure 4-21

\[
T = \frac{3}{2} L_d (L_d I_d + M I_q - L_q I_d)
\]

Rotor Dynamics

\[
\frac{d\omega}{dt} = \frac{T_p - T}{I}
\]

d-axis Circuit

\[
\frac{w_0 L_q I_d}{2}
\]

\[
R_i \rightarrow I_d \rightarrow R \rightarrow V_u
\]

q-axis Circuit

\[
\frac{w_0 L_q (L_d I_d + M I_q)}{2}
\]

\[
R_i \rightarrow I_q \rightarrow R \rightarrow V_u
\]

Field Circuit

\[
V_f \rightarrow I_f \rightarrow R_f
\]

Figure 4-21: Summary of the DQ-frame synchronous generator model.

The generator field excitation \(V_f\) is controlled by proportional-integral feedback of the terminal voltage (120 \(V_{RMS}\) reference), and its mechanical applied torque \(T_{pm}\) is controlled by proportional-integral feedback of the rotor speed (3600 RPM reference). These two feedback loops are described by Figure 4-22. The load resistance \(R_l\) is determined by the state of all connected loads. In this simulation, these are the ECUs, a latrine pump, and an assumed base load of resistance 43.2Ω.

All generator model quantities are tabulated in Table 4.2. Instead of simulating
Figure 4-22: PI feedback control of synchronous terminal voltage and rotor speed.

The interaction of multiple generators, this model approximates generator dispatch both on and off as a smooth change in the total inertia, machine inductances, rotor speed control gains, and generator output impedances as an aggregate. For example, the process of one generator turning on to run in parallel with one other is modeled as a time-linear doubling of the rotor inertia, generation capacity, and rotor speed control gains, and as a time-linear halving of the stator inductances and output impedance. This approximation is not expected to recreate the actual dynamics of generator paralleling, nor does it attempt to recreate the active paralleling and load-sharing controls at play when a generator turns on or off at the BCIL. However, like the generators at the BCIL, it does allow the frequency deviation in response to an ECU state change to depend on the number of generators operating. Also like the BCIL generators, it creates a distinguishable frequency transient when a generator turns on or off.

The generator model has two outputs: electrical frequency $f$ and in-phase seventh harmonic voltage amplitude $P7$. Because it is a synchronous machine model, the electrical frequency is directly proportional to the rotor mechanical speed as described by (4.21).

$$f = \frac{1}{2\pi} \cdot \frac{P}{2} \cdot \omega_m$$

(4.21)
<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>State</td>
<td>1 or 2</td>
</tr>
<tr>
<td>I</td>
<td>Parameter</td>
<td>0.1N kg·m²</td>
</tr>
<tr>
<td>$L_d$</td>
<td>Parameter</td>
<td>3.11/N mH</td>
</tr>
<tr>
<td>$L_f$</td>
<td>Parameter</td>
<td>3 H</td>
</tr>
<tr>
<td>$L_q$</td>
<td>Parameter</td>
<td>3.11/N mH</td>
</tr>
<tr>
<td>M</td>
<td>Parameter</td>
<td>82.7 mH</td>
</tr>
<tr>
<td>P</td>
<td>Parameter</td>
<td>2</td>
</tr>
<tr>
<td>$R_f$</td>
<td>Parameter</td>
<td>5 Ω</td>
</tr>
<tr>
<td>$R_s$</td>
<td>Parameter</td>
<td>0.1N Ω</td>
</tr>
<tr>
<td>$k_{pm}$</td>
<td>Parameter</td>
<td>2N</td>
</tr>
<tr>
<td>$k_{im}$</td>
<td>Parameter</td>
<td>20N</td>
</tr>
<tr>
<td>$k_{pv}$</td>
<td>Parameter</td>
<td>1</td>
</tr>
<tr>
<td>$k_{iv}$</td>
<td>Parameter</td>
<td>10</td>
</tr>
<tr>
<td>$T_m$</td>
<td>Input</td>
<td>N/A</td>
</tr>
<tr>
<td>$V_f$</td>
<td>Input</td>
<td>N/A</td>
</tr>
<tr>
<td>$R_l$</td>
<td>Input</td>
<td>N/A</td>
</tr>
<tr>
<td>$V_f$</td>
<td>Input</td>
<td>N/A</td>
</tr>
<tr>
<td>$i_d$</td>
<td>State</td>
<td>N/A</td>
</tr>
<tr>
<td>$i_f$</td>
<td>State</td>
<td>N/A</td>
</tr>
<tr>
<td>$i_q$</td>
<td>State</td>
<td>N/A</td>
</tr>
<tr>
<td>$\omega_m$</td>
<td>State</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 4.2: DQ Machine Model Parameters
The generator model subsumes the P7 voltage source characteristics and circuit impedances detailed in Section 4.2.1. Output $P7$ is therefore a function of the total load and the number of generators operating which tracks the data shown in Figure 4-12.

4.7.2 Preliminary Simulation Results

For preliminary validation of the autonomous ECU control concept, a camp of eight tents serviced by eight ECUs were simulated for four hours of nighttime operation with a steady environmental temperature of 12°C. The setpoint of the ECUs was set to 21°C. The ECUs operated autonomously and each inferred the number of other ECUs operating their heaters as described in Section 4.4. The pump operated and generated collision transients, which the autonomous ECUs were able to handle using the procedure described in Figure 4-20. The resources available with only one generator operating (no more than four heaters allowed to operate simultaneously) were not sufficient for the ECUs to achieve their setpoint, but the tent temperatures did not deviate far enough from setpoint to trigger the condition described by 4.18. This preliminary result suggests that autonomous ECU operation based on local voltage measurements is feasible, even in the presence of a load which degrades the ability of the ECU controllers to track and identify events of interest.
Figure 4-23: Preliminary simulation results with eight ECUs operated autonomously in the presence of a latrine pump. The dotted red line indicates the average number of ECUs operating during the simulation period.
Chapter 5

Conclusion

This study of coordinated control of environmental control units (ECUs) in an effort to improve the fuel efficiency of forward operating bases (FOBs) with islanded microgrids has: characterized the typical Army FOB microgrid using field measurements from an installed Force Provider module at the Base Camp Integration Laboratory (BCIL) in Devens, MA; developed and deployed a prototype control system that retrofits to the FOB’s existing ECUs and subjects them to central control; formulated two alternative central control schemes that reduce the camp’s peak load while still maintaining thermal control of the serviced compartments; identified and exploited an opportunity to use voltage transients in FOB microgrids to identify remote load events; and formulated an autonomous and network-free method of coordinating ECU operation, which captures the benefits of centralized ECU scheduling without networking requirements. Future work should further explore the possibility of autonomous and network-free ECU coordination by incorporating the principles and methods discussed in this thesis with the hardware developed and installed at the BCIL.
Appendix A

Hardware Documentation

A.1 The F100-60K Environmental Control Unit (ECU)

The F100-60K ECU is the device responsible for maintaining adequate temperatures in the various compartments of a U.S. Army Forward Operating Base.

Figure A-1: Drawing of the ECU with major components and panels indicated [5].

For the typical user, all interactions with the ECU happen at two panels. The first of these is a breaker panel located on the outer face of the Electrical Component Access compartment. It contains resettable overheat, underpressure, and overpressure
circuit breakers. If the ECU operates properly, the operator should not have to touch this panel. The second panel is the Control Access Panel, which contains two dials that are critical to the operation of the ECU. The first dial is a mode selection switch. The switch has four positions, corresponding to the four modes in which the unit can operate: OFF, VENT, COOL, and HEAT. The second dial controls the temperature setpoint of the ECU. The ECU’s setpoint is the temperature that it tries to attain in the room, tent or compartment to which it is attached. The behavior of the unmodified ECU, when it operates normally and hasn’t entered any kind of fault state, can be controlled only through these two dials. Direct control of how much power the ECU consumes, or when it does so, is not possible without some modifications. This appendix will describe the modifications that allowed this project’s exploration of alternative ECU control strategies, which are made entirely within the ECU’s Electrical Component Access compartment. To better understand those modifications, we begin with an overview of the ECU in its unaltered state. The following three sections describe the operation of the unmodified ECU in VENT, HEAT, and COOL mode, and the control circuitry in the Electrical Component Access compartment responsible for it. They make reference to Figure A-2, which depicts the contents of the Electrical Component Access compartment.
Figure A.2: The contents of the F100-60K Electrical Component Access compartment.
Although the contents of the Electrical Component Access compartment may look like an overwhelming mass of wires and components, the bulk of the ECU's function can be understood by observing just a few key circuits of the system at large. These are the circuits associated with mode selection, heater operation, compressor control. Nearly all of this circuitry is accessible at the locations labeled "Terminal Block 1" and "Compressor Controller" in Figure A-2.

A.1.1 Mode Selection Circuitry

The mode selection circuitry is what allows user input via the mode selection dial to control how the ECU operates. The mode selection circuit routes a 24VDC control signal to various places in the ECU, depending on the position of the mode selection switch. The circuitry immediately surrounding the switch is shown in Figure A-3. This schematic demonstrates the essential function of the mode selection switch,

Figure A-3: The mode selection switch circuitry. The switch is a single-pole four-throw rotary switch labeled "S2-1" in the schematic. The common node of the switch is connected to a 24VDC control voltage bus [5]. The circuit has several points of connection to the ECU's Terminal Block 1, which are labeled in the schematic as "TB1-x."

which is labeled as "S2-1." The common node of the switch is connected to two wires in the Electrical Component Access panel: Wire 102B-2 and Wire 601. Wire 102B-2
connects the switch to the second terminal of the ECU’s Terminal Block 1 (TB1-2), which is energized with the ECU’s 24VDC control signal. Wire 601 just routes the control signal to an "On" indicator light.

The switch can connect the 24VDC control voltage to any one of the four points labeled "11 Off," "12 Vent," "13 Heat," and "14 Cool," at a time. Which point is connected determines how the ECU operates. The connection point labeled "11 Off" actually does not connect the control voltage to any additional circuit, so the ECU does nothing when the switch is in the "Off" position.

The connection point labeled "12 Vent" connects the control voltage to three wires: 103A, 103E, and 701. Wire 103A carries the control voltage to the coil of a switch which powers the ECU’s supply fan. If no fault has occurred, this will result in the ECU’s supply fan turning on and circulating air through its compartment. Wire 103E carries the control voltage to a resistive divider circuit, and applies a bias to the control signal that determines the speed of the supply fan. Wire 701 applies power to the ECU’s main thermostat, which measures the temperature of the return air and compares it to the temperature setpoint.

The connection point labeled "13 Heat" connects the control voltage to wires 105(A,B) and to wires 104(A,B,C). Wires 105(A,B) supply power to the contactor that energizes the heater so that, as long as there is no high-heat fault, the thermostat is able to open and close it. Wires 104(A,B,C) connect the control voltage to a switch controlled by the thermostat. In normal operation, the thermostat controls whether or not the control voltage carried by Wires 104(A,B,C) connects to the actuation circuit for the heater contactor, thereby controlling whether or not the heater is energized. In the ECU’s "Override" configuration, which is selected by another switch inside of the Electrical Component Access compartment, the control voltage bypasses the thermostat so that the heater always operates when the protection circuitry connected to Wires 105(A,B) is in its normal state.

The connection point labeled "14 Cool" connects the control voltage to Wires 106(A,B), which carry the control voltage to the heat pump control circuits. These include speed control circuits for the condenser fan, which only operates when the
ECU is in Cool mode, protection breakers that activate based on the pressure measured in the heat pump, and a compressor soft-start system that limits the inrush current when the device first turns on.

There are two additional points of note regarding the mode selection circuit. First observe that the diodes labeled "D1" and "D2" are connected such that, when the mode selection dial is positioned to energize either the Heat or the Cool circuits, the vent circuitry is also energized. It is necessary that the ventilation fan operate any time the heater may be energized, or the heat pump operated, because air flow in either case is critical to the safe operation of the circuit and to the delivery of the desired heating or cooling effects to the connected compartment. Second, note that all of the wires mentioned in this description of the mode selection circuits are accessibly via the ECU’s Terminal Block 1, which is highlighted in Figure A-2. The Vent circuits all connect at TB1-3, the Heat circuits at TB1-4 and TB1-5, and the Cool circuits at TB1-6. These connection points will be important when the time comes to install a controller that can override the ECU’s normal operation.

A.1.2 Heat Mode Operation

The F100-60K can heat its compartment by dissipating heat in a three-phase resistive heater. It energizes the heater by connecting it to the three-phase, 120V_RMS power lines that power the unit. In its unaltered state, the ECU will only ever energize the heater when the mode selection switch on the Control Access Panel is set to HEAT. Beyond this, the logic applied by the unit to decide when to energize the heater is simple: If the heater is NOT energized, and the temperature of its compartment is more than approximately 5°F below the setpoint, then it energizes the heater until the compartment temperature has risen approximately 5°F above the setpoint. The inverse logic also applies: If the heater IS energized, and the temperature of its compartment is more than approximately 5°F above the setpoint, then it de-energizes the heater until the compartment temperature is approximately 5°F below the setpoint.

This control, in addition to a 12s delay imposed when the ECU first enters Heat
mode, is implemented by the circuit shown in A-4. Recall from Wires 104(A,C) in this figure carry a control voltage when the ECU’s mode selection switch is set to the Heat position. The switch at the bottom of the schematic in Figure A-4 is the Override switch mentioned in Section A.1.1. With the Override switch in the position shown, the heater contactor is energized only when both the 12s delay timer and the S350 thermostat switches are closed. The ECU in this case will execute control of the heater described at the beginning of this section. When the Override switch is in the other position, such that nodes five and four are connected, the timer and thermostat switches are bypassed such that the heater is always on—subject to the protective overrides described in Section A.1.1. Having control of the ECU’s mode switch and being able to bypass the timer and thermostat, then, together provide the ability to turn the ECU’s heater on and off at will. Note that the node used by the Override switch to bypass the timer and thermostat is also accessible in the Electronic Component Access compartment (TB1-13).

A.1.3 Cool Mode Operation

In Cool mode, the F100-60K ECU operates a heat pump to reject heat from its compartment. The primary active component of the heat pump is a scroll compressor. The defining characteristic of a scroll compressor is that it can be rapidly loaded and unloaded by actuating a pressure release solenoid. When the solenoid is de-energized,
the compressor operates at its full capacity and draws rated power. When the solenoid is energized the compressor is unloaded because its pressure is released, and it draws far less power. The ECU’s compressor controller controls the rate at which the heat pump rejects heat from the compartment by periodically energizing and de-energizing the unloader solenoid. A simplified picture of the compressor controller from the product manual is shown in Figure A-5. Pictures of the device itself are contained in Figures A-6 and A-7.

![Simplified schematic of the compressor controller from [2]](image)

Figure A-5: Simplified schematic of the compressor controller from [2]. Only components which are active in control of the F100-60K ECU are shown.

The compressor is responsible for managing two components of the compressor: Its contactor, which controls power to the compressor motor; and its unloader solenoid, which allows the controller to control the power of the heat pump as described previously in this section. It can de-energize the compressor by opening the triac at port M2, through which the contactor coil is connected to the controller’s 24VAC control voltage at port L2. Similarly, it can energize and de-energize the unloader solenoid by opening and closing the triac at port U2.
During normal operations in Cool mode, the compressor controller energizes and de-energizes the unloader solenoid at regular intervals. The portion of each interval the compressor spends fully loaded depends on the level of the demand signal voltage at ports C1 and C2. This signal normally comes from the S350 thermostat in Figure A-4, and is higher when the thermostat’s feedback control indicates that more heat rejection is required. Intercepting and substituting this demand signal voltage would be sufficient to control the average power consumption of the compressor, but not to control the exact timing of its energy consumption. To do the latter, direct control must be assumed of at least the unloader solenoid.
The compressor is a far more complex and dynamic system than the resistive heater that the ECU operates in Heat mode, and accordingly the compressor controller performs a broad array of monitoring and protection functions. The compressor measures current to the compressor, pressure within its machinery, and refrigerant temperature in order to quickly identify and react to faults. It can detect and react to eight kinds of faults, which it identifies using the aforementioned measurements. It is not desirable to completely override this controller, because doing so would shift the considerable burden of compressor monitoring and protection onto the overriding controller. It is far preferable to exert direct control of the unloader solenoid and whether or not the compressor is energized, but in such a way that the ECU's compressor controller continues to monitor for faults and retains its ability to quickly react to them. In this project, that is achieved by overriding the compressor controller's unloader solenoid actuation circuit and by substituting the S350 thermostat's demand signal with one that the implanted controller can manipulate. Although this shifts the burden of appropriately manipulating the unloader solenoid during normal operation onto the implanted controller, it leaves the ECU's compressor controller in charge of fault detection and fault handling. As will be described in Section A.2.4, the implanted controller includes circuitry that immediately returns full control of the compressor to the ECU's compressor controller if it detects a fault.

If the compressor controller detects any kind of fault in the operation of the compressor, it closes a normally unused alarm relay (A1, A2) and takes emergency protective actions. The fault states that the compressor controller can detect are enumerated on the back of the device in Figure A-7. It is crucial that the Rev. 2 ECU controller not impede these emergency procedures. Accordingly, the Rev. 2 controller's compressor circuitry includes a feature that automatically returns full control of the compressor to the compressor controller whenever the alarm relay closes. This circuit is described in greater detail in Section A.2.4.
A.1.4 Legacy Mode Operation Rules

When the ECU operates normally with no override from an implanted controller, the only important operating rule to remember is not to turn the ECU directly from Heat mode to Off mode. This can overheat the resistive element and cause the overheat circuit breaker to trip. If the overheat circuit breaker does trip, it is resettable from the outer panel of the Electrical Component Access panel.

A.2 Taking Control

Full control over the mode in which an ECU operates, and when it consumes power in either mode, is a cornerstone of practically evaluating ECU control schemes in the field. As such, it is necessary to have implanted controllers that can modify, bypass, or override the native ECU control circuits described in Section A.1. This section describes two revisions of an implanted controller that retrofit to the F100-60K ECU and subject them to central control. Both revisions are deployed at the BCIL, so it is useful to understand the capabilities, limitations, and configurations of both. Although the two implanted controllers have different capabilities, they have similar hardware and are both based on the same basic plan for assuming control of the ECU’s native control circuitry:

1. Bypass the mode selection switch and use relays on the implanted controller to determine which circuits are energized. This amounts to removing Wire 102B-2 from Terminal TB1-2, and instead plugging the implanted controller into TB1-2 so that it can route the ECU control voltage to TB1-3 (Vent circuit), TB1-4,5 (Heat circuit), and TB1-6 (Cool circuit) independent of the mode selection switch on the Control Access Panel

2. Bypass the heater delay timer and thermostat switch in Heat mode so that the heater can be turned on and off by the implanted controller at will. The Rev.1 controller does not do this directly, but relies on the Override switch inside of the Electrical Component Access compartment being turned on. The Rev. 2
controller does this directly by routing the ECU control voltage to TB1-13 when it wishes to turn the heater on.

3. Override the ECU compressor controller’s management of the compressor’s unloader solenoid in cool mode by disconnecting the solenoid from Port U2 on the compressor controller, and passing it instead through a triac on the implanted controller. Only the Rev.2 controller has the on-board hardware required to accomplish this, but the Rev. 1 controller was designed to allow for the modular addition of similar circuitry.

4. Intercept the S350 thermostat’s demand signal voltage and substitute a control voltage from the implanted controller, so that the ECU’s compressor controller can be manipulated into energizing the compressor. Both revisions of the implanted ECU controller possess this ability, but currently only the Rev. 2 controller exercises it.

A.2.1 Controller Installation: General

Installation of both revisions involves placing an enclosure that contains a DC power supply, a fuse, and the controller itself inside of the ECU’s Electrical Component Access compartment. An example of the placement of the enclosure is provided in Figure A-8. The implanted control enclosure connects electrically to the ECU in three locations: To the power terminals (through a fused connection) so that the enclosure power supply can draw and condition power to provide to the implanted controller; to the ECU compressor controller, for monitoring with the Rev.1 controller and for controlling with the Rev.2 controller; and to the ECU’s Terminal Block 1, where both implanted controller revisions override the ECU’s native mode control circuitry. Zooming in on Terminal Block 1 as in Figure A-9 allows for visual identification of several of the connections both the Rev. 1 and Rev. 2 controller make at Terminal Block 1:

For clarity, the wires that are relevant to ECU installation are highlighted with bold colors in Figure A-9. These connections between the implanted ECU controller
Figure A-8: Rev. 2 controller installed in the Electrical Component Access compartment of the ECU.

Figure A-9: A close-up of the connections made by the implanted controllers to Terminal Block 1.

and the ECU's Terminal Block One are:
A.2.2 ECU Controller: Revision 1

The Rev. 1 ECU controller is installed in ECUs 5-8 at the BCIL. It is capable of controlling the ECU in Heat mode, monitoring ECU operation using its analog-digital converters, and measuring output and return air temperatures with its thermocouple sensors. Figure A-10 gives an overview of the subsystems of the implanted controller, and Figure A-11 gives a bottom view.

Interlock Switches

The Rev. 1 controller is implanted into the ECU with an enclosure that contains three user interlock switches. When the ECU Electrical Component Access compartment is opened completely, the interlock switches are on the face of the implanted enclosure pointing directly away from the compartment. The switches are named "SC1," "SC2," and "SC3," in Figure A-10. In the figure, the switches are oriented as they are...
in the actual installation. That is, when the ECU’s Electrical Component Access compartment panel is fully opened, the switches face outward and are ordered SC1, SC2, SC3 from left to right.

On the inside of the enclosure, each switch has three ports. For each interlock switch, from top to bottom, these connection points are named "1," "2," and "3." The interlock switch’s port 2 is always connected to either 1 or to 3, and never to both. Each switch has two positions: one in which the bat is positioned upwards (called "MIT" position for switches SC1 and SC2), and one in which the bat is positioned downwards (called "BCIL" position for switches SC1 and SC2). In Figure A-10, all switches in the MIT position. When the switch is in the MIT position, ports 2 and 3 are connected. When the switch is in the BCIL position, ports 1 and 2 are connected.

Setting switches SC1 and SC2 to the BCIL position prevents the implanted controller from overriding the ECU’s mode selection switch and its thermostat demand signal, respectively. SC3 functions to prevent the controller from energizing the ECU’s Vent and Heat circuits simultaneously. When it is in the upward position (as shown in Figure A-10), the implanted controller can put the ECU into Heat mode but not into Cool mode. When it is in the downward position, the implanted controller can
put the ECU in Cool mode but not into Heat mode.

**Mode Selection Switches**

The Rev.1 controller uses a set of five electromechanical relays (R1-5 in Figure A-12) to control the operating mode of the ECU. The drive circuitry and configuration of the relays are also displayed in Figure A-12.

![Figure A-12: Schematic view of the Rev.1 ECU controller's mode selection circuitry.](image-url)
K-Type Thermocouple Sensors

The K-Type thermocouple sensors can be used to measure temperature in the range of (-250, 750)°C with a nominal accuracy of ±2°C. The K-type thermocouple circuits in the Rev. 1 implanted controller are packaged Adafruit products called AD8405 (product ID: 1778). The circuits output a voltage that varies approximately linearly with the absolute temperature measured at the location of the thermocouple sensor. This voltage is routed on the Rev.1 controller to terminals B1 and B2. Temperature is resolved by reading the voltage with the implanted controller’s analog to digital converter.

Offset and gain error were observed in the operation of these thermocouples, and in some cases calibration was necessary to achieve the desired precision. In addition to the error margins specified for the AD8495 thermocouple sensor circuit, error can also originate from ADC measurement error. To minimize this it is recommended that only high-precision resistors be used to divide the output voltage of the AD8495 before reading it with the AD8495. It is recommended as best practice for using these devices that they be checked and calibrated against a packaged temperature sensor before testing, and that a packaged temperature sensor also be used during all tests that depend on good temperature resolution for the purposes of corroboration. All temperature data presented in this paper was measured by a packaged third-party temperature sensor.

In the Rev.1 controller, it is important to note that the AD8495 circuit produces an output voltage of 5V when no thermocouple is connected to it. This can damage the analog digital converter, which is rated for 3.3V, so it is strongly recommended that a thermocouple always be connected to the AD8495 when the circuit is energized.

Analog-Digital Converters

The Rev.1 controller uses two eight-channel analog-digital converters. They are rated to read voltages from 0-3.3VDC. The device inputs are interfaced to Terminal B via a voltage divider that multiplies the voltages incident on Terminal B by a factor of 123.
0.122. The voltage divider can be removed or modified where higher-precision, lower-voltage measurements are required. An example of this is reading the voltage output of the AD8495 thermocouple sensor, although caution must be taken because the AD8495 is capable of output voltage that exceeds the ADC rating.

**Digital-Analog Converter (DAC) and Amplifier**

The Rev. 1 controller has two MCP4725 digital-analog converters, the outputs of which are amplified by a two-channel non-inverting amplifier based on the AD8397 integrated circuit. The amplifier is configured for a voltage gain of approximately 3.8. For reasons unknown, the output voltage of the amplifier exhibits a strong nonlinear characteristic and outputs zero volts instead of anything less than approximately 2V. It functions reliably outside of this range, however, and is able to perform all functions required of it for this application.

**Real-Time Clock**

The Sunfounder DS3231 real-time clock is a high-precision timing device that can be used to timestamp data collected by the Rev.1 controller. It has not been fully implemented or used in research as of this writeup.

**A.2.3 Demand Signal Relay**

The demand signal relay is not functional in the Rev.1 controller, due to a layout error that neglected the polarity of its actuator coil.

**A.2.4 ECU Controller: Revision 2**

The Rev. 2 ECU controller is installed in ECUs 1-4 at the BCIL. It is capable of controlling the ECU in Heat mode, assuming full control of the compressor in Cool mode, monitoring ECU operation using its analog-digital converters, and measuring output and return air temperatures with either its thermocouple sensors or the digital
temperature and humidity sensors. Figure A-13 gives an overview of the components of the system.

Interlock Switches

The Rev. 2 controller is implanted into the ECU with an enclosure that contains two user interlock switches. When the ECU Electrical Component Access compartment is opened completely, the interlock switches are on the face of the implanted enclosure pointing left from the perspective looking into the compartment. The switches are named "SC1" and "SC2" in Figure A-13. In the figure, the switches are oriented as they are in the actual installation. That is, when the user is looking at the face of the enclosure that contains the switches, they are ordered SC1, SC2 from left to right.

Figure A-13: Top view of the Rev. 2 Controller.
On the inside of the enclosure, each switch has three ports. For each interlock switch, from top to bottom, these connection points are named "1," "2," and "3." The interlock switch’s port 2 is always connected to either 1 or to 3, and never to both. Each switch has two positions: one in which the upper face of the button is depressed ("MIT" position), and one in which the lower face of the button is depressed ("BCIL" position). In Figure A-10, all switches in the MIT position. When the switch is in the MIT position, ports 2 and 3 are connected. When the switch is in the BCIL position, ports 1 and 2 are connected.

Mode Selection Switches

The Rev.2 controller uses a set of five electromechanical relays (R1-5 in Figure A-15) to control the operating mode of the ECU. The drive circuitry and configuration of the relays are also displayed in Figure A-15.
K-Type Thermocouple Sensors

The K-Type thermocouple sensors can be used to measure temperature in the range of (-250, 750)°C with a nominal accuracy of ±2°C. The K-type thermocouple circuits in the Rev. 2 implanted controller are layout and component-for-component identical with the Adafruit product called AD8405 (product ID: 1778). The circuits output a voltage that varies approximately linearly with the absolute temperature measured at the location of the thermocouple sensor. This voltage is routed on the Rev.2 controller directly to channels 7 and 8 of the controller’s on-board analog-digital converter.

Offset and gain error were observed in the operation of these thermocouples, and in some cases calibration was necessary to achieve the desired precision. In addition to the error margins specified for the AD8495 thermocouple sensor circuit, error can also originate from ADC measurement error. It is recommended as best practice for using these devices that they be checked and calibrated against a packaged temperature sensor before testing, and that a packaged temperature sensor also be used during all tests that depend on good temperature resolution for the purposes of corroboration. All temperature data presented in this paper were measured by a packaged third-party temperature sensor.

Analog-Digital Converters

The Rev.2 controller uses one eight-channel analog-digital converter. It is rated to read voltages from 0-3.3VDC. Device input channels 1-6 are interfaced to Terminals E3 and E4 via a voltage divider that multiplies the voltages incident on Terminals E3 and E4 by a factor of 0.067. Input channels 7 and 8 are connected directly to the controller’s on-board thermocouple sensor circuits, which have been altered from the Rev.1 controller so that they no longer output voltages in excess of the ADC ratings.

Digital-Analog Converter (DAC) and Amplifier

The Rev.2 controller has two MCP4725 digital-analog converters, the outputs of which are amplified by a two-channel non-inverting amplifier based on the AD8397 inte-
grated circuit. The amplifier is configured for a voltage gain of approximately 3.8. For reasons unknown, the output voltage of the amplifier exhibits a strong nonlinear characteristic and would output zero volts instead of anything less than approximately 2V. It functions reliably outside of this range, however, and is able to perform all functions required of it for this application. Critically, it can reliably provide a high voltage signal to the compressor controller when necessary to allow direct control of the compressor solenoid.

**Real-Time Clock**

The Sunfounder DS3231 real-time clock is a high-precision timing device that can be used to timestamp data collected by the Rev.1 controller. It has not been fully implemented or used in research as of this writeup.

**A.2.5 Demand Signal Relay**

The demand signal relay (R6) is a signal-level relay that has less current capacity than the mode-switching relays but also a smaller contact impedance. It is used to control the source from which the ECU’s compressor controller measures the demand signal that determines its operating mode.

**A.3 Temperature and Humidity Sensor**

The Rev.2 controller can interface via I2C with a temperature a humidity sensor, which plugs into the controller via ethernet cable at either of Ports E2 or E3.

**A.4 Compressor State Monitor**

To enable embedded ECU controllers to monitor the regular operation of the compressor controller, a half-wave rectifier and a low-pass filter with resistive voltage division can be connected to the compressor controller in the manner specified by Figure A-18.
A.5 Installation Wiring Schematics

Wiring schematics are included in this section to assist with the installation of future controller revisions, or the expansion of existing installations.
Figure A-15: Schematic view of the Rev.2 ECU controller's mode selection circuitry.

Figure A-16: Top view of the i2c-enabled temperature and humidity sensor.
Figure A-17: Bottom view of the i2c-enabled temperature and humidity sensor.

Figure A-18: A simple RC filter can be used to indicate when the compressor solenoid is and is not energized with a DC voltage level. This allows monitoring to be accomplished with the analog-digital circuitry on the embedded ECU controllers.
Figure A-19: Wiring guide for the mode control portion of the Rev.1 controller.
Figure A-20: Wiring guide for the mode control portion of the Rev. 2 controller.
Figure A-21: Wiring guide for the compressor control portion of the Rev. 2 controller.
Appendix B

Software Documentation

This thesis was enabled by a body of Python and MATLAB software that can be separated into seven broad categories:

1. Raspberry Pi ECU control
2. MATLAB ECU control
3. MATLAB temperature data analysis/display
4. MATLAB tent thermal modeling
5. MATLAB FOB simulation
6. MATLAB voltage data analysis/display
7. MATLAB voltage transient identification

The member scripts of each category are described in this Appendix.

B.1 Raspberry Pi ECU Control

All of the scripts that enabled the Raspberry Pi-based controller of this thesis to execute ECU control were written for interpretation by a Python 3 interpreter. In general, they contain instructions that make the Raspberry Pi communicate with
a master controller using the Modbus/TCP protocol and interface with its various connected devices to execute ECU control as an effective control system. The file structure (B.1) of the Raspberry Pi ECU control suite reflects this separation of responsibilities into communication, device interfacing, and control system implementation.

Main Directory

- ECU_Controller.py
- Comms
  - modbus.py
  - regbank.py
  - __init__.py
- Systems
  - Compensator.py
  - Feedback.py
  - ModeSwitch.py
  - __init__.py
- Devices
  - ADC.py
  - CLK.py
  - DAC.py
  - IO.py
  - Relay.py
  - TempSense.py
  - __init__.py
  - __addr__.py
Of the files shown in B.1, 10 are modules that perform the essential functions of the suite: Communications modules modbus.py and regbank.py; Device modules ADC.py, CLK.py, DAC.py, Relay.py, and TempSense.py; and Systems modules Compensator.py, Feedback.py, and ModeSwitch.py. These modules interact in a hierarchical fashion. The Communications (Comms) modules receive instructions to make the Raspberry Pi perform an action, and pass that instruction to the appropriate Systems module. The Systems modules break that requested action down into a series of device operations, evaluate its validity in the context of the ECU’s current operating state, and instruct the Device modules to perform the corresponding operations if the action is valid; The Device modules interface with the Raspberry Pi’s various control and monitoring devices to execute the operations that, together, comprise the requested action. The scripts, along with a brief summary of each, are listed below:

B.1.1 ECU_Controller.py

The script ECU_Controller.py is the high-level module that binds all of the others together and configures them based on the setup of the Raspberry Pi Controller.

```python
#!/usr/bin/env python3

### Import Devices, Systems, and Communications protocols
import Devices #Scripts to interface with the MCP3008 (ADC), DS3231 (Clock), MCP4725 (DAC),
import Systems
import Comms.regbank as r #Import the REGistor BANK. Reading/writing to these registers is how the central controller interfaces with the RPi
import Comms.modbus as mb #Import the modbus functions that allow the RPi to receive/interpret/encode/send modbus messages
###

# Initialize controller’s IP address
```

137
IP = '192.168.10.84' #MUST match the Raspberry Pi's own static IP address

(https://www.modmypi.com/blog/how-to-give-your-raspberry-pi-a-static-ip-address-update)

#Initialize controller's ADC. Arguments to the initialization function are the scales applied by each channel's voltage divider (ch1 divider multiplies input voltage by .0667). NONE of these are "settable..." the values reflect gains fixed in hardware
ADC1 = Devices.ADC.ADC_MCP3008([.0667, .0667, .0667, .0667, .0667, .0667, 1, 1])

#Initialize controller's DACs. Arguments are (Amplifier Gain, I2C address) NEITHER of these are "settable..." the values reflect gains and addresses fixed in hardware
DAC1 = Devices.DAC.DAC_MCP4725(3.8, 0x61)
DAC2 = Devices.DAC.DAC_MCP4725(3.8, 0x60)

#Initialize controller's temperature and/or humidity sensors
TC1 = Devices.TempSense.Thermocouple_AD8495(ADC1, 7) #TC1 is an AD8495 thermocouple, whose output is read by channel 7 of ADC1
TC2 = Devices.TempSense.Thermocouple_AD8495(ADC1, 6) #TC2 is an AD8495 thermocouple sensor, whose output is read by channel 6 of ADC1
THS1 = Devices.TempSense.Sensor_HDC1010(0x41)
#THS2 = Devices.TempSense.Sensor_HDC1010(0x43)

#Initialize the controller's feedback control machinery
feedback = Systems.feedback_node(THS1) #Temperature measured by Temperature/Humidity Sensor #1 (THS1) is the feedback variable
compensator = Systems.Compensator() #Default args: (setpoint=-1 deg. F, sample_time=1s, op_mode='Vent', comp_mode='I', d0=0, throttle=10, int_gain = 0.5)
MS = Systems.ModeSwitch(ADC1, DAC1) #
# Initialize the controller’s bank of registers for Modbus interfacing

```plaintext
r1001 = r.read_reg(MS, 0)  # ECU National Stock Number (NSN)
    first group (four digits)

r1002 = r.read_reg(MS, 0)  # NSN second group (two digits)

r1003 = r.read_reg(MS, 0)  # NSN third group (three digits)

r1004 = r.read_reg(MS, 0)  # NSN fourth group (four digits)
```

```plaintext
r2001 = r.read_reg(MS, -1)  # RESERVED for reading
    evaporator inlet temperature

r2002 = r.read_reg(MS, -1)  # RESERVED for reading
    evaporator coil temperature

r2003 = r.mode_check_reg(MS)  # Indicates the operating mode
    of the ECU

r2004 = r.read_reg(MS, -1)  # RESERVED for tracking
    operational hours

r2005 = r.read_reg(MS, -1)  # RESERVED for tracking
    compressor high-pressure cutoff status

r2006 = r.read_reg(MS, -1)  # RESERVED for tracking
    compressor low-pressure cutoff

r2007 = r.setpoint_check_reg(compensator)  # Indicates the current
    setpoint of the ECU (currently only works in MIT mode)

r2008 = r.MIT_BCIL_check_reg(MS)  # Indicates whether ECU is
    under local or remote control

r2009 = r.compressor_check_reg(MS)  # Indicates operating state of
    the compressor

r2010 = r.feedback_reg(feedback)  # Indicates return air temperature

r2011 = r.read_reg(MS, -1)#tempcheck_reg(THS2)  # Indicates output air
    temperature... DEACTIVATED and replaced with a value register because
    THS2 is not initialized

r2012 = r.duty_check_reg(feedback, compensator)  # Indicates ECU’s requested
    duty cycle

r2013 = r.humiditycheck_reg(THS1)  # Indicates return air humidity
```
r2014 = r.read_reg(MS, -1)  # humidity check reg (THS2)  # Indicates output air humidity... DEACTIVATED and replaced with a value register because THS2 is not initialized

r2015 = r.heater_check_reg(MS)  # Returns 1 if heater is on
   if heater is off... IF the ADC channel assigned to monitor the heater is actually connected... otherwise this returns nonsense (probably 0)

r2016 = r.compressor_check_reg(MS)  # Returns 1 if the compressor is energized, 0 if the compressor is off... IF in MIT mode OR configured to monitor in BCIL mode

r2017 = r.fanspeed_check_reg(MS)  # Returns a value corresponding to the vent fan speed setpoint... IF the ADC channel assigned to monitor it is connected.

r2018 = r.solenoid_check_reg(MS)  # Returns

r3001 = r.mode_ctrl_reg(MS, compensator)  # Controls ECU operating mode

r3002 = r.MIT_BCIL_ctrl_reg(MS)  # Controls ECU local/remote control state

r3003 = r.compressor_ctrlSEL_reg(MS)  # Controls compressor scroll local/remote control state

r3004 = r.compressor_ctrl_reg(MS)  # Controls compressor loading/unloading (remote control mode)

r4001 = r.setpoint_ctrl_reg(compensator)  # Sets the ECU temperature setpoint (remote control mode)

r4002 = r.sampletime_ctrl_reg(compensator)  # Sets the sample time used for temperature feedback

r4003 = r.int_gain_ctrl_reg(compensator)  # Sets the compensator integral gain

r4004 = r.throttle_ctrl_reg(compensator)  # Sets the compensator throttling range (controls proportional gain)

r4005 = r.compensation_ctrl_reg(compensator)  # Sets compensation mode of compensator (proportional (0), integral (1), or proportional-integral
r4006 = r.op_mode_ctrl_reg(compensator)  #Sets the operating mode of the compensator (vent (1), cool (2), heat (3))

#Put the registers just initialized into a dictionary, so that they can be easily indexed and accessed.
#reg_dict is what the Modbus script will interact with when it gets read/write requests

if __name__ == '__main__':
    host1 = mb.host(IP, reg_dict)
    try:
        #Tell the Modbus script to begin listening for modbus messages over ethernet
        host1.make_server()
    finally:
        #Reset the state of all Raspberry Pi I/O pins when the ECU controller shuts down
        Devices.ADC.GPIO.cleanup()

B.1.2 modbus.py

The module modbus.py is responsible for instantiating and operating a server that receives commands from the master controller. These commands are sent over a net-
work connection and follow the Modbus/TCP protocol. The instructions are always sent as a read/write instruction to a particular register. When modbus.py forwards that read/write command to the module regbank.py, it is translated into an instruction to be given to one of the Systems modules.

```python
#!/usr/bin/env python3

# If you're not an expert in the TCP/IP stack... neither am I.
# Go to this website (https://pypi.org/project/uModbus/) to download
# the necessary packages if the import incantations aren't working.
# Make sure that the packages you download go to the proper file location
#/usr/local/lib/python3.8/dist-packages or
#/usr/local/lib/python3.8/site-packages

### BEGIN MODBUS INCANTATIONS
import logging
from socketserver import TCPServer
# from collections import defaultdict
from umodbus import conf
from umodbus.server.tcp import get_server, RequestHandler
from umodbus.utils import log_to_stream
# from regbank import GPIO, reg_dict
# print(regbank.__file__)

log_to_stream(level=logging.DEBUG)

# data_store = defaultdict(int)

conf.SIGNED_VALUES = True

TCPServer.allow_reuse_address = True
### END MODBUS INCANTATIONS
```
class host(object):
    def __init__(self, addr, reg_dict):
        self.addr = addr
        self.app = get_server(TCPServer, (self.addr, 502), RequestHandler)
        self.reg_dict = reg_dict

    def make_server(self):
        @self.app.route(slave_ids=[1], function_codes=[3, 4],
                        addresses=list(range(0, 10000)))
        def read_data_store(slave_id, function_code, address):
            #
            print("Received function code 3!")
            print('received read command')
            return self.reg_dict[address].val

        @self.app.route(slave_ids=[1], function_codes=[6, 16],
                        addresses=list(range(0, 10000)))
        def write_data_store(slave_id, function_code, address, value):
            #
            print("Received function code 6!")
            self.reg_dict[address].val = value

        try:
            self.app.serve_forever()
        finally:
            self.app.shutdown()
            self.app.server_close()

        #if __name__ == '__main__':
        # host1 = host('192.168.10.83')
        # try:
        # host.app.serve_forever()
# finally:
# host.app.shutdown()
# host.app.server_close()
# GPIO.cleanup()

B.1.3 regbank.py

This is the interface between the communications modules and the Systems modules. regbank.py is responsible for translating a register read/write command into an instruction to be given to one of the Systems modules, such as taking control of the ECU or changing its operating mode.

#!/usr/bin/env python3

# DATA STRUCTURING SECTION
# Registers of the read_reg clas cannot be written to

class read_reg(object):
    def __init__(self, controller, val=-1):
        self.controller = controller
        self._val = val

    @property
def val(self):
        return int(self._val)

def val(self, val):
    pass

# Registers of the write_reg class CAN be written to

class write_reg(object):
    def __init__(self, controller, reg_r=-1):
        pass
#If the write register has a read register that mirrors its actions, then it is specified here
self.reg_r = reg_r
self.controller = controller

@property
def val(self):
    return -1

@val.setter
def val(self, write_val=0):
    #Some write registers, when written to, do an action that
corresponds to the written valueself.
#self.execute() is redefined for different kinds of registers so
that they do the correct actions
#when written to.
sel.execute(write_val)
if self.reg_r != -1:
    #If the write register has a read register that mirrors its
    actions, then update it after executing the write action!
sel.reg_r._val = write_val

def execute(self, write_val):
    pass

#Control the operating mode of the ECU (off, vent, heat, or cool)
class mode_ctrl_reg(write_reg):
    def __init__(self, controller, compensator, reg_r=-1):
        self.controller = controller
        self.compensator = compensator
        self.reg_r = reg_r
def execute(self, mode):
    # You can only change the ECU operating mode if the controller has
    # overridden the ECU and put it in "MIT mode"
    print('checking to see if in MIT mode...')
    if self.controller.MIT_Control.state == 1:
        print('in MIT mode... executing mode change!',
              self.controller.checkmode() )
        if mode == 0:
            self.controller.off()
        elif mode == 1:
            self.controller.vent()
        elif mode == 2:
            self.controller.cool()
        elif mode == 3:
            self.controller.heat()
        print('done executing mode change!')

    # Check the operating mode of the ECU (0-off, 1-vent, 2-cool, 3-heat)
    class mode_check_reg(read_reg):
        @property
        def val(self):
            if self.controller.checkmode() == 'Off':
                return 0
            elif self.controller.checkmode() == 'Vent':
                return 1
            elif self.controller.checkmode() == 'Cool':
                return 2
            elif self.controller.checkmode() == 'Heat':
                return 3
#Check whether the ECU heater is on or off

class heater_check_reg(read_reg):
    @property
    def val(self):
        return self.controller.checkheater()

#Check the state of the compressor (loaded or unloaded)

class solenoid_check_reg(read_reg): 
    @property
    def val(self):
        if self.controller.MIT_Sol Override.state == 1:
            if self.controller.MIT_Sol_Unloader.state == 1: 
                #TO DO: add an additional logic condition (ECU Compressor controller NOT in alarm state), or change the test to directly measure whether or not solenoid is energized.
                return 0 #Solenoid is energized, compressor is unloaded
            else:
                return 1 #Solenoid is deenergized, compressor is loaded
        else:
            return -1 #ECU is under BCIL control, compressor state is unknown. TO DO: develop hardware to directly measure compressor state via ADC, or measure state as reported by compressor controller via status light

#Check whether the compressor is energized or not

class compressor_check_reg(read_reg): 
    @property
    def val(self):
        if self.controller.mode == 'Cool':

147
return 1 #Solenoid is deenergized, compressor is loaded
else:
    return 0 #ECU is under BCIL control, compressor state is unknown. TO DO: develop hardware to directly measure compressor state via ADC, or measure state as reported by compressor controller via status light

#Override control of the ECU compressor unloader solenoid
class compressor_ctrlSEL_reg(write_reg):
    def execute(self, state):
        if state == 1:
            self.controller.sol_takecontrol()
        else:
            self.controller.sol_returncontrol()

#If controller has overridden ECU control of compressor solenoid, writing to this register exercises that control
class compressor_ctrl_reg(write_reg):
    def execute(self, state):
        if state == 1:
            self.controller.sol_load() #Deenergize compressor solenoid, load compressor
        else:
            self.controller.solUnload() #Energize compressor solenoid, unload compressor

#Check whether ECU is under MIT or BCIL control
class MIT_BCIL_check_reg(read_reg):
    @property
def val(self):
    return self.controller.MITControl.state()

# Control whether ECU is under MIT or BCIL control
class MIT_BCIL_ctrl_reg(write_reg):
    def execute(self, MITctl):
        if MITctl == 1:
            self.controller.takecontrol()
        else:
            self.controller.returncontrol()

# Check a system or environmental temperature
class tempcheck_reg(read_reg):
    def __init__(self, sensor):
        self.sensor = sensor

    @property
    def val(self):
        return int(10*self.sensor.read_temp())

# Check a system or environmental humidity
class humiditycheck_reg(tempcheck_reg):
    @property
    def val(self):
        if self.sensor.name == 'HDC1010':
            return int(10*self.sensor.read_humidity())

# Checks the Raspberry Pi's OWN setpoint, NOT that of the ECU control panel
class setpoint_check_reg(read_reg):
    def __init__(self, compensator):
        self.compensator = compensator
        self.setpoint = self.compensator.setpoint

@property
def val(self):
    return self.compensator.setpoint

@val.setter
def val(self, value):
    pass

# Change the temperature setpoint the Raspberry Pi will attempt to achieve
class setpoint_ctrl_reg(write_reg):
    def __init__(self, compensator, reg_r=-1):
        self.compensator = compensator
        self.reg_r = reg_r

    def execute(self, write_val):
        self.compensator.setpoint = write_val

# The master controller decides how much time is between samples... changing
# this value simply notifies the Raspberry Pi (local controller) what that
# interval
# is... Useful primarily if you are doing integral feedback and have
# specified a
# gain.
class sampletime_ctrl_reg(write_reg):
    def __init__(self, compensator, reg_r=-1):
self.compensator = compensator
self.reg_r = reg_r

def execute(self, write_val):
    if write_val > 0:
        self.compensator.sample_time = write_val

#The feedback_node measures the temperature used for feedback
class feedback_reg(read_reg):
    def __init__(self, feedback_node):
        self.feedback_node = feedback_node

@property
def val(self):
    temp = self.feedback_node.collect_data()
    #Multiply by 10 to achieve 0.1 resolution... must divide at the receiving end!
    #This is because modbus only accommodates integer values
    return int(10*temp)

#Reading this register causes the Raspberry pi to recalculate its duty cycle requests
#based on the feedback temperatures it has read since the last query
class duty_check_reg(read_reg):
    def __init__(self, feedback_node, compensator):
        self.compensator = compensator
        self.feedback_node = feedback_node

@property
def val(self):
print('attempting to get a duty cycle')
duty_cycle =
    self.compensator.compute(self.feedback_node.transmit_data('average'),
    self.feedback_node.samplecount)
self.feedback_node.reset_node()
print('duty cycle is', duty_cycle)
return int(duty_cycle)

#Change the integral feedback used to determine the Raspberry Pi's duty
cycle request
class int.gain.ctrl-reg(write_reg):
    def __init__(self, compensator, reg_r=-1):
        self.compensator = compensator
        self.reg_r = reg_r

    def execute(self, int.gain):
        self.compensator.int-gain = int.gain/1000

#If using proportional feedback, writing to this register changes the
throttling range
#of the feedback... LOWER throttling range means HIGHER proportional gain
class throttlectrl_reg(write_reg):
    def __init__(self, compensator, reg_r=-1):
        self.compensator = compensator
        self.reg_r = reg_r

    def execute(self, throttle):
        self.compensator.throttle = throttle/10
# Change the compensation mode... proportional, integral, or proportional-integral!

class compensation_ctrl_reg(write_reg):
    def __init__(self, compensator, reg_r=-1):
        self.compensator = compensator
        self.reg_r = reg_r

    def execute(self, comp_mode):
        self.compensator.reset()

        if comp_mode == 0:
            self.compensator.comp_mode = 'P'
        elif comp_mode == 1:
            self.compensator.comp_mode = 'I'
        elif comp_mode == 2:
            print('Setting compensation mode to PI')
            self.compensator.comp_mode = 'PI'

# Change the operating mode of the compensator so that it calculates its duty cycle appropriately to whether it is in Heat or Cool mode

class opmode_ctrlreg(write_reg):
    def __init__(self, compensator, reg_r=-1):
        self.compensator = compensator
        self.reg_r = reg_r

    def execute(self, op_mode):
        if op_mode == 1:
            self.compensator.op_mode = 'V'
        elif op_mode == 2:
            self.compensator.op_mode = 'C'
elif op_mode == 3:
    print('received op_mode HEAT command')
    self.compensator.op_mode = 'H'

#Check the fan speed
class fanspeed_check_reg(read_reg):
    @property
def val(self):
        fan_setpoint = self.controller.fancheckspeed()
        return int(fan_setpoint*10)

B.1.4 Compensator.py

When the command sent to the Raspberry Pi is related to the way in which the operation of the ECU heater or compressor depends on the ECU’s setpoint and feedback temperature, one of the Systems modules that regbank.py sends instructions to is Compensator.py. This module controls how the Raspberry Pi converts temperature errors over time into a prescribed duty cycle for the ECU in whichever mode it is operating.

#!/usr/bin/env python3
class Compensator(object):
    def __init__(self, setpoint=-1, sample_time=1, op_mode='Vent',
                 comp_mode='I', d0=0, throttle=10, int_gain = 0.5):

        #Initialize temperature setpoint
        self.setpoint = setpoint
        #Initialize sample time for integral control
        self.sample_time = sample_time
        #Initialize starting duty cycle of heater/cooler
self.d0  = d0
self.duty = 0

#Initialize throttling range for proportional control
self.throttle = throttle

#Initialize integral gain for integral control ( (% duty
cycle)/(degree error * minute) )
self.int_gain = int_gain

#Initialize integral term of duty cycle to 0 (for operation with
integral feedback)
self.duty_I = self.d0

#Initializes the operating mode of the ECU (heating, cooling, or
neither?)
self.op_mode = op_mode

#Initializes the compensation mode of the compensator
self.comp_mode = comp_mode

#Initialize ECU operating mode
if self.op_mode == 'Heat':
    self.op_mode = 'H'
elif self.op_mode == 'Cool':
    self.op_mode = 'C'
else:
    self.op_mode = 'V'

#Initialize compensation mode
if comp_mode == 'P':
    self.comp_mode = 'P'
elif comp_mode == 'I':
    self.comp_mode = 'I'
elif comp_mode == 'PI':
    self.comp_mode = 'PI'
elif comp_mode == 'PID':

self.comp_mode == 'PID'
else:
    self.comp_mode == 'OL'

def prop_calc(self, err):
    duty_P = 100*err/self.throttle

    #Don't let the integral portion of the gain go above 100 or below -100

    if duty_P > 100:
        duty_P = 100
    elif duty_P < -100:
        duty_P = -100
    print('calculating duty_P!')
    print(err)
    print(self.throttle)
    return duty_P

def int_calc(self, err, samplecount):
    #Integrate (int_gain*sample_time*error) to yield new duty cycle
    self.duty_I = self.duty_I +
        self.int_gain*self.sample_time*samplecount*err

    #Don't let the integral portion of the gain go above 100 or below -100
    if self.duty_I > 100:
        self.duty_I = 100
    if self.duty_I < -100:
        self.duty_I = -100

    return self.duty_I

def compute(self, feedback, samplecount):
if feedback != -1001:
    #Calculate error signal based on ECU operating mode
    #Positive error increases control effort
    if self.op_mode == 'H':
        err = self.setpoint - feedback
    elif self.op_mode == 'C':
        err = feedback - self.setpoint
    else:
        print('not in heat or cool operating mode!')
        err = 0

    #Calculate new duty cycle based on compensation mode
    if self.comp_mode == 'P':
        print('conducting proportional feedback')
        self.duty = self.prop_calc(err)
    elif self.comp_mode == 'I':
        self.duty = self.int_calc(err, samplecount)
    elif self.comp_mode == 'PI':
        print('in PI feedback mode! computing...')
        self.duty = self.int_calc(err, samplecount) +
                    self.prop_calc(err)

    ###TO DO: implement PID and OL duty cycle computations!

    if self.duty > 100:
        self.duty = 100
    elif self.duty < 0:
        self.duty = 0

    return self.duty

def reset(self):
B.1.5 Feedback.py

Feedback.py is responsible for controlling the method by which the ECU’s setpoint and measured feedback temperature are converted into error signals to be handled by Compensator.py.

```python
#!/usr/bin/env python3

class feedback_node(object):
    def __init__(self, sensor):
        self.sensor = sensor
        self.samplecount = 0
        self.temp = 0
        self.temp_agg = 0
        self.temp_array = [0 for x in range(1000)]
        self.dataflag = 0

    def collect_data(self):
        self.temp_last = self.temp
        self.temp = self.sensor.read_temp()
        if self.temp == -1:
            self.temp = self.temp_last
        self.temp_agg = self.temp_agg + self.temp
        self.temp_array[self.samplecount] = self.temp
        self.samplecount = self.samplecount + 1
        self.dataflag = 1
        return self.temp

    def transmit_data(self, query='average'):
        if self.dataflag == 1:
```

158
if query == 'latest':
    val = self.temp  # Returns last temperature sample taken
elif query == 'average':
    val = int(self.temp_agg/self.samplecount)  # Returns average of temperatures sampled over the previous period
elif query == 'array':
    val = self.temp_array[0:self.samplecount-1]  # Returns whole array of temperatures sampled over the previous period
else:
    val = -1000  # Indicates invalid argument
return val
else:
    return -1001  # Indicates attempt to read when no data is available

def reset_node(self):
    print('c')
    self.temp = 0
    print('d')
    self.temp_agg = 0
    print('e')
    self.temp_array = [0 for x in range(1000)]
    print('f')
    self.samplecount = 0
    print('g')
    self.dataflag = 0
    print('h')
    return
B.1.6 ModeSwitch.py

ModeSwitch.py is the workhorse module for changes to the ECU’s operating mode. Any time an instruction requires the ECU to change its operating mode, activate or deactivate its heater, energize or de-energize the compressor unloader solenoid, or monitor some aspect of the ECU’s operation, regbank.py sends instructions to ModeSwitch.py. ModeSwitch.py is also responsible for a series of protocols which prevent poor operation of the ECU. For example, it will not obey a request that requires turning the ECU directly from heat mode to vent mode, in an effort to prevent overheat errors.

```python
#!/usr/bin/env python3

from Devices.Relay import relay
import time

#Use board location of pins (J8-XX)
#XX represents the address of the Raspberry Pi (RPi) GPIO pin that controls the...
MIT_CTRL = 32 #Relay that determines whether the RPi is in control of mode selection
MIT_DMND = 12 #Relay that determines whether the RPi is in control of the demand signal sent to the compressor controller
VENT = 31 #Relay that determines whether or not the ECU is in vent mode, IF the RPi is in control of mode selection
HC_ENABLE = 22 #Relay that allows the RPi to put the ECU in heat or vent mode, IF the RPi is in control of mode selection
HC_SELECT = 18 #Relay that determines whether the ECU is in heat mode or cool mode, IF the RPi is allowed to do so by HC_ENABLE and MIT_CTRL relays
```
SOLOVERRIDE = 37  # Relay that determines whether the RPi is in control of the compressor solenoid
SOLUNLOADER = 36  # Relay that determines whether the compressor unloader solenoid is energized, IF the RPi is controlling it
HEATOVERRIDE = 16  # Relay that determines whether or not the RPi overrides the ECU's delay and thermostat heater controls

# X represents the ADC channel that...
VENT_MON_CH = 0  # Detects whether the ECU is in Vent mode
COOL_MON_CH = 1  # Detects whether the ECU is in Cool mode
HEAT_MON_CH = 2  # Detects whether the ECU is in Heat mode
HEAT_OVR_MON_CH = 3  # Detects whether the ECU heater is currently on
FANSPEED_MON_CH = 4  # Monitors the ECU ventilation fan speed setpoint

class ModeSwitch(object):
    def __init__(self, adc, dac):
        self.MIT_Control = relay(MIT_CTRL)
        self.MIT_Demand = relay(MIT_DMND)

        self.MIT_Vent = relay(VENT)

        self.MIT_HCEnable = relay(HC_ENABLE)
        self.MIT_HCSelect = relay(HC_SELECT)

        self.MIT_Sol_Override = relay(SOL_OVERRIDE)
        self.MIT_Sol_Unloader = relay(SOL_UNLOADER)

        self.MIT_Heat_Override = relay(HEAT_OVERRIDE)

        self.mode_monitor = adc
self.demand_signal = dac

if self.mode_monitor.readadc(HEAT_MON_CH) > 20:
    self.mode = 'Heat'
elif self.mode_monitor.readadc(COOL_MON_CH) > 20:
    self.mode = 'Cool'
elif self.mode_monitor.readadc(VENT_MON_CH) > 20:
    self.mode = 'Vent'
else:
    self.mode = 'Off'

def off(self):
    print('entered function off()')
    if self.MIT_Control.state == 1:
        if self.mode == 'Off':
            return
        elif self.mode == 'Vent':
            self.MIT_Vent.turnoff()
        elif self.mode == 'Heat':
            self.MIT_Vent.turnon()
            self.MIT_HEnable.turnoff()  # Do not allow direct turnoff from heat mode
        elif self.mode == 'Cool':
            if self.MIT_Sol_Override.state == 1:  # Do some stuff to transition from COOL to VENT
                self.MIT_Sol_Unloader.turnon()
                time.sleep(0.5)
                self.MIT_Sol_Unloader.turnoff()
                time.sleep(0.5)
            if self.MIT_Demand.state == 1:
                self.demand_signal.set_voltage(0)  # Demand goes to 0V
                self.MIT_HEnable.turnoff()
time.sleep(1)
self.MIT_Sol_Override.turnoff()

else:
    if self.MIT_Demand.state == 1:
        self.demand_signal.set_voltage(0)  # Demand goes to 0V
        self.MIT_HCEnable.turnoff()
    self.mode = 'Off'
    return
else:
    return

def vent(self):
    print('entered function vent()')
    if self.MIT_Control.state == 1:  # IF operating in MIT mode
        self.MIT_Vent.turnon()
        if self.mode == 'Heat':
            print('detected in heat mode')
            self.MIT_HCEnable.turnoff()
        elif self.mode == 'Cool':
            if self.MIT_Sol_Override.state == 1:  # Do some stuff to transition from COOL to VENT
                self.MIT_Sol_Unloader.turnon()
                #
                time.sleep(.5)
                self.MIT_Sol_Unloader.turnoff()
                time.sleep(0.5)
            if self.MIT_Demand.state == 1:
                self.demand_signal.set_voltage(0)  # Demand goes to 0V
                self.MIT_HCEnable.turnoff()
                time.sleep(1)
                self.MIT_Sol_Override.turnoff()
            else:
if self.MITDemand.state == 1:
    self.demand_signal.set_voltage(0)  # Demand goes to OV
    self.MITHCEnable.turnoff()
    self.mode = 'Vent'
    return
else:
    print('ECU in BCIL mode')
    return  # IF operating in BCIL mode, cannot change operating mode

def heat(self):
    print('entered function heat()')
    if self.MITControl.state == 1:  # IF operating in MIT mode
        if self.mode == 'Heat':
            return
        elif self.mode == 'Cool':
            return  # Do some stuff to transition from COOL to HEAT
        elif (self.mode == 'Vent') | (self.mode == 'Off'):
            self.MITHCSelect.turnoff()  # Select HEAT with Heat-Cool select relay
            self.MITHCEnable.turnon()  # Turn on Heat-Cool enable relay to activate HEAT mode
            self.MITVent.turnoff()
            self.mode = 'Heat'
            return
        else:
            print('ECU in BCIL mode')
            return  # IF operating in BCIL mode, cannot change operating mode
def cool(self):
    print('entered function cool()')
    if self.MIT_Contrl.state == 1:  # IF operating in MIT mode
        if self.mode == 'Heat':
            return  # Do some stuff to transition from HEAT to COOL
        elif self.mode == 'Cool':
            return
        elif (self.mode == 'Vent') | (self.mode == 'Off'):
            self.MIT_HCSelect.turnon()  # Select COOL with Heat-Cool select relay
            if self.MIT_Demand.state == 1:
                self.demand_signal.set_voltage(5)  # Demand goes to 5V
                self.MIT_HCEnable.turnon()  # Turn on Heat-Cool enable relay to activate COOL mode
                self.MIT_Vent.turnoff()
                if self.MIT_Sol_Override.state == 1:
                    self.MIT_Sol_Unloader.turnon()
                    time.sleep(.1)
                    self.MIT_Sol_Unloader.turnoff()
            self.mode = 'Cool'
            return
        else:
            return  # IF operating in BCIL mode, cannot change operating mode

def checkmode(self):
    if self.MIT_Control.state == 0:
        # TO DO: Use ADC to check BCIL operating mode and change
        self.mode accordingly
        if self.mode_monitor.readadc(HEAT_MON_CH) > 20:
self.mode = 'Heat'
elif self.mode Monitor.readadc(COOL_MON_CH) > 20:
    self.mode = 'Cool'
elif self.mode Monitor.readadc(VENT_MON_CH) > 20:
    self.mode = 'Vent'
else:
    self.mode = 'Off'
a = str(self.mode)
return a

elif self.MIT_Control.state == 1:
    print('in MIT mode... returning operating mode info...')
a = str(self.mode)
return a

def checkheater(self):
    if self.mode Monitor.readadc(HEAT_OVR_MON_CH) > 20:
        return 1
    else:
        return 0

def takecontrol(self):
    if self.MIT_Control.state == 1:
        print('Already in MIT mode!')
        return #Already in control!
    else:
        if (self.checkmode() == 'Cool') | (self.checkmode() == 'Heat'):
            print('Can't take control because in Heat or Cool mode!')
            return #Don't change control directly from cool or heat mode!
elif (self.checkmode() == 'Off') | (self.checkmode() == 'Vent'):
        print('Taking control...')
sel.Off()
def returncontrol(self):
    if self.MIT_Control.state == 0:
        print('Already in BCIL mode!')
        return #Already returned control!
    else:
        if (self.checkmode() == 'Cool') | (self.checkmode() == 'Heat'):
            print('mode interlock!')
            return #Don’t change control directly from cool or heat mode!
        elif (self.checkmode() == 'Off') | (self.checkmode() == 'Vent'):
            print('executing off command')
            self.MIT_Control.turnoff()
            self.MIT_Demand.turnoff()
            self.MIT_Heat_Override.turnoff()

    def sol_takecontrol(self):
        if self.MIT_Control.state == 0:
            return #Don’t override solenoid in BCIL mode!
        elif self.MIT_Control.state == 1:
            # if (self.checkmode() == 'Cool') | (self.checkmode() == 'Heat'):
            #     return #Don’t change control directly from cool or heat mode!
            # elif (self.checkmode() == 'Off') | (self.checkmode() == 'Vent'):
            #     if self.MIT_SolOverride.state == 1:
            #         return #Already in control of solenoid!
            #     elif self.MIT_SolOverride.state == 0:
            #         self.MIT_Sol_Unloader.turnoff()
            #         self.MIT_SolOverride.turnon()
            if self.MIT_SolOverride.state == 1:
                return #Already in control of solenoid!
            elif self.MIT_SolOverride.state == 0:
                self.MIT_SolOverride.state = 0:
def sol_returncontrol(self):
    if (self.checkmode() == 'Cool') | (self.checkmode() == 'Heat'):
        return #Don't change control directly from cool or heat mode!
    elif (self.checkmode() == 'Vent') | (self.checkmode() == 'Off'):
        if self.MIT_Sol_OVERRIDE.state == 0:
            return #Already returned control!
        elif self.MIT_Sol_OVERRIDE.state == 1:
            self.MIT_Sol_Unloader.turnoff()
            time.sleep(0.02) #Allow triac output to turn off before opening electromechanical relay
            self.MIT_Sol_OVERRIDE.turnoff()

def sol_load(self):
    if (self.MIT_Control.state == 1) & (self.checkmode() == 'Cool') & (self.MIT_Sol_OVERRIDE.state == 1):
        if self.MIT_Sol_Unloader.state == 0:
            return #Already loaded!
        elif self.MIT_Sol_Unloader.state == 1:
            self.MIT_Sol_Unloader.turnoff()
    else:
        return #Not in a state where it makes sense to load/unload compressor

def sol_unload(self):
    if (self.MIT_Control.state == 1) & (self.checkmode() == 'Cool') & (self.MIT_Sol_OVERRIDE.state == 1):
        if self.MIT_Sol_Unloader.state == 1:
            return #Already unloaded!
elif self.MIT_Sol_Unloader.state == 0:
    self.MIT_Sol_Unloader.turnon()
else:
    return  # Not in a state where it makes sense to load/unload

def fan_checkspeed(self):
    print('polling ADC for fan speed setpoint...')
    return self.mode_monitor.readadc(FANSPEED_MON_CH)

B.1.7 ADC.py

ADC.py is responsible for interfacing with the Raspberry Pi's analog digital converter. It is called any time a function requires the measurement of a voltage in the ECU or on the Raspberry Pi's control circuitry.

#!/usr/bin/env python3

import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BOARD)

class ADC_MCP3008(object):
    def __init__(self, scale, CS=26, MISO=21, MOSI=19, CLK=23):
        self.MISO = MISO
        self.MOSI = MOSI
        self.CLK = CLK
        self.CS = CS
        self.scale = scale

        GPIO.setup(self.MISO, GPIO.IN)
GPIO.setup(self.MOSI, GPIO.OUT)
GPIO.setup(self.CLK, GPIO.OUT)
GPIO.setup(self.CS, GPIO.OUT)

# Thanks to Adafruit for this portion of code!
def readadc(self, adcnum=0):
    if ((adcnum > 7) or (adcnum < 0)):
        return -1

    GPIO.output(self.CS, True)
    GPIO.output(self.CLK, False)  # start clock low
    GPIO.output(self.CS, False)  # bring CS low

    commandout = adcnum
    commandout |= 0x18  # start bit + single-ended bit
    commandout <<= 3  # we only need to send 5 bits here

    for i in range(5):
        if (commandout & 0x80):
            GPIO.output(self.MOSI, True)
        else:
            GPIO.output(self.MOSI, False)
        commandout <<= 1
        GPIO.output(self.CLK, True)
        GPIO.output(self.CLK, False)

    adcout = 0  # read in one empty bit, one null bit and 10 ADC bits

    for i in range(12):
        GPIO.output(self.CLK, True)
        GPIO.output(self.CLK, False)
        adcout <<= 1
        if (GPIO.input(self.MISO)):
            adcout |= 0x1
```python
GPIO.output(self.CS, True)

adcout >>= 1  # first bit is 'null' so drop it
voltage = adcout*3.3/(1023*self.scale[adcnum])
print('Voltage read by ADC:', voltage)
return voltage
```

B.1.8 CLK.py

CLK.py is responsible for interfacing with the Raspberry Pi's real time clock circuit. This module has not yet been used, but is functional.

```python
#!/usr/bin/env python3

#NOTE: This code is from FaBoPlatform. Original can be found at
#github.com/FaBoPlatform/FaBoRTC-PCF2129-Python/blob/master/FaBoRTC_PCF2129/PCF2129.py

class CLK_PCF2129(object):
    def __init__(self, addr, twi=1, smbus=None, time=None):
        if smbus is None:
            import smbus
        if time is None:
            import time

        self._addr = addr
        self._bus = smbus.SMBus(twi)
        self.time = time

        self.CONTROL_REG = (0x00)
        self.CONTROL_12_24 = (0x04)
        self.SECONDS = (0x03)
```
self.MINUTES = (0x04)
self.HOURS = (0x05)
self.DAYS = (0x06)
self.WEEKDAYS = (0x07)
self.MONTHS = (0x08)
self.YEARS = (0x09)

## Configure Device

def configure(self):
    self.set24mode()

## Get Seconds from RTC
# @return seconds
def getSeconds(self):
    data = self._bus.read_byte_data(self._addr, self.SECONDS)
    return self.bcdToDec(data)

## Set Seconds to RTC
# @param [in] seconds seconds
def setSeconds(self, seconds):
    if (seconds > 59) or (seconds < 0):
        seconds = 0

    data = self.decToBcd(seconds) + 0x80
    self._bus.write_byte_data(self._addr, self.SECONDS, data)

## Get Minutes from RTC
# @return minutes
def getMinutes(self):
    data = self._bus.read_byte_data(self._addr, self.MINUTES)
    return self.bcdToDec(data)
## Set Minutes to RTC

```python
# @param [in] minutes minutes
def setMinutes(self, minutes):
    if (minutes > 59) or (minutes < 0):
        minutes = 0

    data = self.decToBcd(minutes)
    self._bus.write_byte_data(self._addr, self.MINUTES, data)
```

## Get Hours from RTC

```python
# @return hours
def getHours(self):
    data = self._bus.read_byte_data(self._addr, self.HOURS)
    return self.bcdToDec(data)
```

## Set Hours to RTC

```python
# @param [in] hours hours
def setHours(self, hours):
    self.set24mode()
    if (hours > 23) or (hours < 0):
        hours = 0

    data = self.decToBcd(hours)
    self._bus.write_byte_data(self._addr, self.HOURS, data)
```

## Get Days from RTC

```python
# @return days
def getDays(self):
    data = self._bus.read_byte_data(self._addr, self.DAYS)
    return self.bcdToDec(data)
```

## Set Days to RTC
# @param [in] days days

def setDays(self, days):
    if (days > 31) or (days < 1):
        days = 1

    data = self.decToBcd(days)
    self._bus.write_byte_data(self._addr, self.DAYS, data)

## Get Weekdays from RTC
## @return weekdays

def getWeekdays(self):
    data = self._bus.read_byte_data(self._addr, self.WEEKDAYS)
    return self.bcdToDec(data)

## Set Weekdays to RTC
## @param [in] weekdays weekdays

def setWeekdays(self, weekdays):
    if (weekdays > 6) or (weekdays < 0):
        weekdays = 0

    data = self.decToBcd(weekdays)
    self._bus.write_byte_data(self._addr, self.WEEKDAYS, data)

## Get Months from RTC
## @return months

def getMonths(self):
    data = self._bus.read_byte_data(self._addr, self.MONTHS)
    return self.bcdToDec(data)

## Set Months to RTC
## @param [in] months months

def setMonths(self, months):
    data = self._bus.write_byte_data(self._addr, self.MONTHS, data)
def setMonths(self, months):
    if (months > 12) or (months < 1):
        months = 1
    data = self.decToBcd(months)
    self._bus.write_byte_data(self._addr, self.MONTHS, data)

## Get Years from RTC
# @return years
def getYears(self):
    data = self._bus.read_byte_data(self._addr, self.YEARS)
    return self.bcdToDec(data)

## Set Years to RTC
# @param [in] years years
def setYears(self, years):
    if (years > 99) and (years < 0):
        years = 0
    data = self.decToBcd(years)
    self._bus.write_byte_data(self._addr, self.YEARS, data)

## Read from RTC
# @retval year Read Years
# @retval month Read Months
# @retval day Read Days
# @retval hour Read Hours
# @retval minute Read Minutes
# @retval second Read Seconds
def now(self):
    data = self._bus.read_i2c_block_data(self._addr, self.SECONDS, 7)

    seconds = self.bcdToDec(data[0])
    minutes = self.bcdToDec(data[1])

    seconds = 0
    minutes = 0
hours = self.bcdToDec(data[2])
days = self.bcdToDec(data[3])

# blank read weekdays
months = self.bcdToDec(data[5])
years = self.bcdToDec(data[6]) +2000

return {'year':years, 'month':months, 'day':days, 'hour':hours,
        'minute':minutes, 'second':seconds}

## Set to RTC

# @param [in] DateTime DateTime
def setDate(self, years, months, days, hours, minutes, seconds):
data = [self.decToBcd(seconds) | 0x80,
       self.decToBcd(minutes),
       self.decToBcd(hours),
       self.decToBcd(days),
       0x00,
       self.decToBcd(months),
       self.decToBcd(years-2000)]

self._bus.write_i2c_block_data(self.addr,self.SECONDS, data)

## Set to 12 hour mode
def set12mode(self):
    ctrl = self.readCtrl() | CONTROL_12_24
    self.writeCtrl(ctrl)

## Set to 24 hour mode
def set24mode(self):
    ctrl = self.readCtrl() & ~(CONTROL_12_24)
    self.writeCtrl(ctrl)
## BCD to DEC
# @param [in] value BCD value
# @param [out] value DEC value
def bcdToDec(self, value):
    return (((value-(value%16)) / 16 * 10) + (value % 16))

## DEC to BCD
# @param [in] value DEC value
# @param [out] value BCD value
def decToBcd(self, value):
    return int(((value-value%10) / 10 * 16) + (value % 10))

## Read Control Register
# @param [out] data register data
def readCtrl(self):
    return self._bus.read_byte_data(self._addr, CONTROL_REG)

## Write Control Register
# @param [in] data register data
def writeCtrl(self, data):
    self._bus.write_byte_data(self._addr, CONTROL_REG, data)

B.1.9 DAC.py

DAC.py is responsible for interfacing with the Raspberry Pi's two digital to analog converters.

#!/usr/bin/env python3

#Default I2C address:
DEFAULT_ADDRESS = 0x61
RAIL_VOLTAGE  = 5

class DAC_MCP4725(object):
    
    """Base functionality for MCP4725 digital to analog converter."""

    def __init__(self, amplifier_gain=1, address=DEFAULT_ADDRESS, i2c=None, 
                 **kwargs):
        """Create an instance of the MCP4725 DAC."""
        if i2c is None:
            import Adafruit_GPIO.I2C as I2C
            i2c = I2C

        self._device = i2c.get_i2c_device(address, **kwargs)
        self.WRITEDAC = 0x40
        self.WRITEDACEEPROM = 0x60
        self.amplifier_gain = amplifier_gain
        self.RAIL_VOLTAGE = RAIL_VOLTAGE
        self.set_voltage(0)

    def set_voltage(self, value, persist=False):
        """Set the output voltage to specified value. Value is a 12-bit number 
        (0-4095) that is used to calculate the output voltage from:

            Vout = (VDD*value)/4096

        I.e. the output voltage is the VDD reference scaled by value/4096. 
        If persist is true it will save the voltage value in EEPROM so it 
        continues after reset (default is false, no persistence).

        """

        # Clamp value to an unsigned 12-bit value.
        value = int(value*4096/(self.RAIL_VOLTAGE*self.amplifier_gain))

        if value > 4095:
value = 4095
if value < 0:
    value = 0
#logging.debug('Setting value to {0:04}'.format(value))
# Generate the register bytes and send them.
# See datasheet figure 6-2:
# https://www.adafruit.com/datasheets/mcp4725.pdf
reg_data = [(value >> 4) & 0xFF, (value << 4) & 0x0F]
if persist:
    self._device.writeList(self.WRITEDACEEPROM, reg_data)
else:
    self._device.writeList(self.WRITEDAC, reg_data)

B.1.10 Relay.py

Relay.py is responsible for operating the Raspberry Pi's onboard control relays. This includes its electromechanical mode selection and demand signal routing relays, as well as the solid state relay which modulates the ECU's compressor unloader solenoid.

```python
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BOARD)  # Define GPIO pins in terms of their physical locations; GPIO.BCM uses BCM numbering instead

class relay(object):
    def __init__(self, pin):
        GPIO.setup(pin, GPIO.OUT)  # Set the relay's drive pin to output mode
        self.pin = pin
        self.turnoff()
    def turnon(self):
        GPIO.output(self.pin, True)  # Switch relay contact to normally-open position
        self.state = 1
```

179
def turnoff(self):
    GPIO.output(self.pin, False)  # Revert relay to normally-closed position
    self.state = 0

B.1.11 TempSense.py

TempSense.py is responsible for interfacing with all of the Raspberry Pi's temperature sensing devices. An instance of the TempSense.py object can be configured as a thermocouple, or as a digital temperature humidity sensor. Any future temperature or humidity sensing devices can easily be added into this script.

#!/usr/bin/env python3

class Thermocouple_AD8495(object):
    def __init__(self, adc, channel):
        self.adc = adc
        self.channel = channel
        self.name = 'Thermocouple'

    def read_temp(self):
        print(self.channel)
        tc_output = self.adc.readadc(self.channel)
        tempc = (tc_output - 1.25) / .005
        tempf = tempc * 1.8 + 32
        print('temperature measured by sensor:', tempf)
        return tempf

class Sensor_HDC1010(object):
    def __init__(self, addr, twi=1, smbus=None, time=None):
        if smbus is None:
            import smbus

180
print(smbus.__file__)

if time is None:
    import time

self.name = 'HDC1010'
self._addr = addr
self._bus = smbus.SMBus(twi)
self.time = time
self.temp = 0

self.TEMP_REG = 0x00
self.HUMIDITY_REG = 0x01
self.CONFIG_REG = 0x02
self.MANUFACTUREID_REG = 0xFE
self.DEVICEID_REG = 0xFF
self.SERIALIDHIGH_REG = 0xFB
self.SERIALIDMID_REG = 0xFC
self.SERIALIDBOT_REG = 0xFD

self.CONFIG_RST_BIT = 0x8000
self.CONFIG_HEATER_ENABLE = 0x2000
self.CONFIG_ACQUISITION_MODE = 0x1000
self.CONFIG_BATTERY_STATUS = 0x8000
self.CONFIG_TEMPERATURE_RESOLUTION = 0x4000
self.CONFIG_HUMIDITY_RESOLUTION_HBIT = 0x0200
self.CONFIG_HUMIDITY_RESOLUTION_LBIT = 0x0400

self.CONFIG_TEMPERATURE_RESOLUTION_14BIT = 0x0000
self.CONFIG_TEMPERATURE_RESOLUTION_11BIT = 0x0400

self.CONFIG_HUMIDITY_RESOLUTION_14BIT = 0x0000
self.CONFIG_HUMIDITY_RESOLUTION_11BIT = 0x0100
self.CONFIG_HUMIDITY_RESOLUTION_8BIT = (0x0200)

config = self.CONFIG_ACQUISITION_MODE
self._bus.write_byte_data(self._addr, self.CONFIG_REG, config>>8)

def read_temp(self):
    attempt = 0
    success = 0
    while attempt <= 5:
        try:
            self._bus.write_byte(self._addr, self.TEMP_REG)
            self.time.sleep(0.020)

            data0 = self._bus.read_byte(self._addr)
            print(data0)
            success = 1
            break

        except IOError:
            attempt = attempt + 1
            print('failed an attempt to read temperature')
            self.time.sleep(.05)

    if success == 1:
        self.temp = (data0 * 256.0)
        tempc = (self.temp / 65536.0) * 165.0 - 40
        tempf = tempc*1.8 + 32.0
    else:
        tempf = -1
print(round(1.7))
print(round(1.4))
return round(tempf)

def read_humidity(self):
    attempt = 0
    success = 0
    while attempt <= 3:
        try:
            self._bus.write_byte(self._addr, self.HUMIDITY_REG)
            self.time.sleep(0.020)
            data0 = self._bus.read_byte(self._addr)
            success = 1
            humidity = (data0 * 256.0)
            humidity = (humidity / 65536.0) * 100.0
            break
        except IOError:
            attempt = attempt + 1
            print('failed an attempt to read humidity')
            self.time.sleep(0.1)
            if success == 1:
                return humidity
            else:
                return -1001

def read_config(self):
    self._bus.write_byte(self._addr, self.CONFIG_REG)
    self.time.sleep(0.020)
data0 = self._bus.read_byte(self._addr)
data1 = self._bus.read_byte(self._addr)

return data0

def config_resolution(self, temp_res, humidity_res):
    if temp_res == 0:
        self._config(self.CONFIGREG, 10, 0)
    elif temp_res == 1:
        self._config(self.CONFIGREG, 10, 1)

    if humidity_res == 0:  # 14-bit
        self._config(self.CONFIGREG, 9, 0)
        self._config(self.CONFIGREG, 8, 0)
    elif humidity_res == 1:  # 11-bit
        self._config(self.CONFIGREG, 9, 0)
        self._config(self.CONFIGREG, 8, 1)
    elif humidity_res == 2:  # 8-bit
        self._config(self.CONFIGREG, 9, 1)
        self._config(self.CONFIGREG, 8, 0)

    #for debugging only
    # self._bus.write_byte(self._addr, self.CONFIG_REG)
    # data0 = self._bus.read_byte(self._addr)
    # data1 = self._bus.read_byte(self._addr)
    # print(data0, data1)

def _config(self, reg, bit, val):
    adj = 0x0001 << bit
    config = self.read_config()
if val == 0:
    adj = ~adj
    new_config = (config<<8) & adj
else:
    new_config = (config<<8) | adj

self._bus.write_byte_data(self._addr, reg, new_config>>8)

def set_humidity_res(self, res):
    config = self.read_config()
    config = (config<<8 & ~0x0300) | res
    self._bus.write_byte_data(self._addr, self.CONFIGREG, config>>8)
    return

def set_temp_res(self, res):
    config = self.read_config()
    config = (config<<8 & ~0x4000) | res
    self._bus.write_byte_data(self._addr, self.CONFIGREG, config>>8)

def read_battery(self):
    config = self.read_config()
    config = config<<8 & ~self.CONFIG_HEATER_ENABLE
    if config == 0:
        return True
    else:
        return False
    return 0

def read_manufactureID(self):
    data = self._bus.read_i2c_block_data(self._addr,
B.2 MATLAB ECU Control

The MATLAB ECU control suite allows any personal computer with MATLAB R2016a (other versions not tested) to interface with the embedded Raspberry Pi ECU controllers and send them instructions via Modbus/TCP. At the heart of all interactions with the embedded Raspberry Pis is a Modbus/TCP transaction, but executing these transactions is a tedious and unintuitive process. To make the task of ECU
controllable, the MATLAB ECU control suite adds a few layers of abstraction. There are six essential software modules in the ECU control suite:

- Control_ECU.s.m
- ctrl_ECU.m
- collect_data.m
- format_array.m
- scheduler_func.m
- send_command.m

The module that operates at the highest level of abstraction is Control_ECU.s.m. During normal operation of the ECUs, this should be the only module with which a user has to interact. It allows the user to define a sequence of periods of centralized ECU operation, either in Heat or Cool mode, by specifying parameters that include the number of ECUs controlled, the temperature setpoint, the sample interval, and the maximum number of ECUs allowed to operate simultaneously.

Control_ECU.s.m assigns a data structure to each controlled ECU, which is populated with essential information about an ECU pertinent to control and monitoring.

An example of this is included below:

```matlab
1 ECU_2 = struct('Tent', 2,...
2 'IP', '192.168.10.82',...  
3 'Action', 'Control',...
4 'Handles', 0,...
5 'Return_Device', 'T/H Sensor',...
6 'Output_Device', 'T/H Sensor',...
7 'Mode_Monitor', 'Y',...
8 'Compressor_Monitor', 'N');
```

The data contained in ECU#2's data structure indicates the tent to which it is assigned, the IP address at which it can be contacted, whether its action is to control or
just to monitor the ECU, what kind of device it uses to measure the return and output air temperature and/or humidity, whether its operating mode should be recorded as a data sequence, and whether or not the embedded Raspberry Pi controller should monitor the operation of the ECU’s compressor. Each ECU being controlled has a data structure similar to this one, and the information contained in it affects how each module interacts with it.

Control_ECUs.m commands ECU action using plain language commands, using ctrl_ECU.m and send_command.m to work out how to encode the command as a message that the embedded Raspberry Pi will understand. For example, to set ECU#7 in Heat mode, Control_ECUs.m would send the following command:

```matlab
ctrl_ECU(ECU(7).Handles, 'set operating mode', 'heat');
```

In receipt of this command, the module ctrl_ECU.m would call the module send_command.m as follows:

```matlab
send_command([1, 0, 6, 1, 6, 3001, 3], ECU(7).Handles);
```

which is a series of instructions that cause the send_command module to encode the appropriate instruction according to the Modbus/TCP protocol and send it to the correct embedded Raspberry Pi. In addition to simplifying the process of commanding ECUs to take action, the ECU control suite also greatly simplifies the process of data collection from the ECUs. Instrumental in this are the functions format_array.m and collect_data.m. format_array.m generates a labeled data array whose entries depend on the configuration of each ECU as indicated by its data structure in Control_ECUs.m, and collect_data.m functions as a macro which iteratively calls the ctrl_ecu.m module to collect information for every entry in the data array for every timestep.

The final element of the ECU Control Suite is scheduler_func.m. This function schedules ECU operation times according to the algorithm described in Section 3.2.2, which the Control_ECUs.m module executes as described above.
```matlab
B.2.1 Control_ECU.m

1 %%% Clears and Closes
2 clear all;
3 clf;
4 close all;
5
6
7 %%% Specify file in which to save test data
8 filename = ''; %Note: For this to work, create a file with
this name in the directory specified in the "Save Test
  Data" portion of this script.
9 Date = 'SEP 07 2017';
10
11 %%% Specify Test Parameters
12 mode = 'cool';
13 mode = lower(mode);
14
15 setpoints = [58]; %Series of setpoints
16 temp_duration = [4]; %Duration of each
  setpoint in HOURS
17 temp_duration = temp_duration.*3600; %Convert setpoint
duration to SECONDS
18
19 decision_period = .5; %Period (MINUTES)
20 decision_period = decision_period*60; %Convert decision
  period to SECONDS
```
datapoints_per_decision_period = 30;           %Must be integer-valued

seconds_per_datapoint  = decision_period /
                         datapoints_per_decision_period - 0.5;

datapoints_per_setpoint = datapoints_per_decision_period.*
                        temp_duration./decision_period;

setpoint_shift_indices = cumsum(datapoints_per_setpoint);

max_simultaneous = 3;                              %Max. ECUs allowed to
                 run simultaneously

filename = 'C:\Users\2016\Dropbox (MIT)\BCIL\Code\MATLAB\Lab Testing\Summer 2017\18JUN17_Test';       %Filename (.xlsx-type) for data
storage

% Instantiate ECUs
ECU_1 = struct ('Tent', 1,...
                'IP', '192.168.10.81',...
                'Action', 'Control',...
                'Handles', 0,...
                'Return_Device', 'T/H Sensor',...
                'Output_Device', 'T/H Sensor',...
                'Mode_Monitor', 'Y',...
                'Compressor_Monitor', 'N');

ECU_2 = struct ('Tent', 2,...
                'IP', '192.168.10.82',...
'Action', 'Control', ...
'Handles', 0, ...
'Return_Device', 'T/H Sensor', ...
'Output_Device', 'T/H Sensor', ...
'Mode_Monitor', 'Y', ...
'Compressor_Monitor', 'N');

ECU_3 = struct ('Tent', 3, ...
 'IP', '192.168.10.83', ...
 'Action', 'Control', ...
 'Handles', 0, ...
 'Return_Device', 'T/H Sensor', ...
 'Output_Device', 'T/H Sensor', ...
 'Mode_Monitor', 'Y', ...
 'Compressor_Monitor', 'N');

ECU_4 = struct ('Tent', 4, ...
 'IP', '192.168.10.84', ...
 'Action', 'Control', ...
 'Handles', 0, ...
 'Return_Device', 'T/H Sensor', ...
 'Output_Device', 'T/H Sensor', ...
 'Mode_Monitor', 'Y', ...
 'Compressor_Monitor', 'N');

ECU_5 = struct ('Tent', 5, ...
 'IP', '192.168.10.85', ...
 'Action', 'Monitor', ...
 'Handles', 0, ...
 'Return_Device', 'Thermocouple', ...)
'Output_Device', 'Thermocouple', ...
'Mode_Monitor', 'N', ...
'Compressor_Monitor', 'Y');

ECU_6 = struct('Tent', 6, ...
  'IP', '192.168.10.86', ...
  'Action', 'Monitor', ...
  'Handles', 0, ...
  'Return_Device', 'Thermocouple', ...
  'Output_Device', 'Thermocouple', ...
  'Mode_Monitor', 'N', ...
  'Compressor_Monitor', 'Y');

ECU_7 = struct('Tent', 7, ...
  'IP', '192.168.10.87', ...
  'Action', 'Monitor', ...
  'Handles', 0, ...
  'Return_Device', 'Thermocouple', ...
  'Output_Device', 'Thermocouple', ...
  'Mode_Monitor', 'N', ...
  'Compressor_Monitor', 'Y');

ECU_8 = struct('Tent', 8, ...
  'IP', '192.168.10.88', ...
  'Action', 'Monitor', ...
  'Handles', 0, ...
  'Return_Device', 'Thermocouple', ...
  'Output_Device', 'Thermocouple', ...
  'Mode_Monitor', 'N', ...
  'Compressor_Monitor', 'Y');
ECU_8 = struct ('Tent', 8,...
% 'IP', '192.168.10.88',...
% 'Action', 'Bypass',...
% 'Handles', 0,...
% 'Return_Device', 'Thermocouple',...
% 'Output_Device', 'Thermocouple',...
% 'Mode_Monitor', 'Y',...
% 'Mode', -1*ones(datapoints_per_decision_period),...
% 'Return_Temp', -1*ones(datapoints_per_decision_period)
%
% 'Output_Temp', -1*ones(datapoints_per_decision_period)
%
% 'Return_Humidity', -1*ones(datapoints_per_decision_period),...
% 'Output_Humidity', -1*ones(datapoints_per_decision_period),...
% 'Compressor>Loading', -1*ones(datapoints_per_decision_period));

ECU = [ECU_1, ECU_2, ECU_3, ECU_4, ECU_5, ECU_6, ECU_7, ECU_8 ];

[data_array, format_descriptor] = format_array(ECU, sum(datapoints_per_setpoint));

for i = 1:length(ECU)
    if ~strcmp(ECU(i).Action, 'Bypass')
        ECU(i).Handles = tcpip_init({ECU(i).IP});
    end
end
% Initialize ECU controllers
sp = 1;
controlled_ECUs = [];
for i = 1:length(ECU)
    if strcmp(ECU(i).Action, 'Control')
        ctrl_ECU(ECU(i).Handles, 'set sample period',
                 decision_period/datapoints_per_decision_period);
        ctrl_ECU(ECU(i).Handles, 'set temperature setpoint',
                 setpoints(sp));
        ctrl_ECU(ECU(i).Handles, 'set throttling range', 20);
        ctrl_ECU(ECU(i).Handles, 'set integral gain', .02); %
        if strcmpi(mode, 'heat')
            ctrl_ECU(ECU(i).Handles, 'set operating mode', 'heat');
        elseif strcmpi(mode, 'cool')
            ctrl_ECU(ECU(i).Handles, 'set operating mode', 'cool');
        end
        ctrl_ECU(ECU(i).Handles, 'MIT mode', 1);
    end
    controlled_ECUs = [controlled_ECUs, ECU(i).Tent];
end

for i = 1:length(ECU)
if strcmp(ECU(i).Action, 'Control')
    ctrl_ECU(ECU(i).Handles, 'vent', 1);
end
end

pause(10);

%%% Initialize duty cycles and a run schedule for the ECUs
%%% Query duty cycles for each controlled ECU
duty_transmit = -ones(1,8);
setpoint = setpoints(1);
duty = zeros(1, length(controlled_ECUs));

try
    for q = 1:length(controlled_ECUs)
        disp('querying duty cycles for the FIRST time...');
        duty(q) = ctrl_ECU(ECU(controlled_ECUs(q)).Handles, 'query duty cycle', 1);
        if duty(q) > 50
            duty(q) = 50;
        end
    end
    duty_transmit(1, controlled_ECUs(q)) = duty(q);
    if strcmp(mode, 'cool') && (duty(q) < 5)
duty(q) = 0;
ctrl_ECU(ECU(controlled_ECUs(q)).Handles, 'vent', 1);
tvent = tic;

%Switch ECU over to cool mode if duty cycle increases beyond 10%
%AND more than 2 minutes & 10 seconds have elapsed
elseif strcmp(mode, 'cool') && strcmp(ctrl_ECU(ECU(
    controlled_ECUs(q)).Handles, 'check mode', 1), 'vent') && toc(tvent) > 130
    ctrl_ECU(ECU(controlled_ECUs(q)).Handles, 'cool', 1);
    ctrl_ECU(ECU(controlled_ECUs(q)).Handles, 'unload compressor', 1);
end

%Clip duty cycles proportionally to satisfy max simultaneous %run requirement
if sum(duty/100) > max_simultaneous
    duty = duty*(1-(sum(duty/100)-max_simultaneous)/sum(duty/100));
    duty = duty - mod(duty,100/
datapoints_per_decision_period);
end

%Create a new run schedule for the next decision period
schedule = scheduler_func(duty, datapoints_per_decision_period, zeros(1, length(controlled_ECUs)));  

%Reset the datapoint counter to initiate a new decision period
datapoints_this_decision_period = 1;

%% Run
for i = 1:length(data_array)

  %Turn ECUs on and off according to the computed run schedule
  for q = 1:length(controlled_ECUs)
    if schedule(datapoints_this_decision_period, q) == 1
      if strcmp(mode, 'heat')
        ctrl_ECU(ECU(controlled_ECUs(q)).Handles, 'heat', 1);
      elseif strcmp(mode, 'cool')
        ctrl_ECU(ECU(controlled_ECUs(q)).Handles, 'load compressor', 1);
      end
    else
      if strcmp(mode, 'heat')
        ctrl_ECU(ECU(controlled_ECUs(q)).Handles, 'vent', 1);
      elseif strcmp(mode, 'cool')
        ctrl_ECU(ECU(controlled_ECUs(q)).Handles, 'unload compressor', 1);
      end

  end
end
end
end

%Take measurements
newrow = collect_data(ECU, format_descriptor, 
    setpoint, duty_transmit);
data_array(i,:) = newrow;

disp(newrow(1:5));

%Begin a record of how much time setback is caused by 
mode-changes
stoppage_time = 0;

%Update schedule at the end of the decision period
if datapoints_this_decision_period == 
    datapoints_per_decision_period 
    %Query duty cycles for each controlled ECU
for q = 1:length(controlled_ECUs)
    disp('querying duty cycles...');
    duty(q) = ctrl_ECU(ECU(controlled_ECUs(q)). 
        Handles, 'query duty cycle', 1);

    if duty(q) > 50 
        duty(q) = 50;
    end

    duty_transmit(1, controlled_ECUs(q)) = duty(q 
    );
% Change ECU over to vent mode if < 10% duty cycle is requested
if strcmp(mode, 'cool') && (duty(q) < 5)
duty(q) = 0;
ctrl_ECU(ECU(controlled_ECUs(q)).Handles,
    'vent', 1);
stoppage_time = stoppage_time + 1.5;

% Switch ECU over to cool mode if duty cycle increases beyond 10%
elseif strcmp(mode, 'cool') && strcmp(
    ctrl_ECU(ECU(controlled_ECUs(q)).Handles,
        'check mode', 1), 'vent')
    ctrl_ECU(ECU(controlled_ECUs(q)).Handles,
        'cool', 1);
    stoppage_time = stoppage_time + 4;
end

% Clip duty cycles proportionally to satisfy max simultaneous
% run requirement
if sum(duty/100) > max_simultaneous
    duty = duty*(1-(sum(duty/100)-
        max_simultaneous)/sum(duty/100));
duty = duty - mod(duty,100/
    datapoints_per_decision_period);
end
%Create a new run schedule for the next decision period
schedule = scheduler_func(duty,
    datapoints_per_decision_period, zeros(1,
    length(controlled_ECU)));  

%Reset the datapoint counter to initiate a new decision period
datapoints_this_decision_period = 1;  
else
    datapoints_this_decision_period =
    datapoints_this_decision_period + 1;
end  

%Update temperature setpoints when necessary
if sum(setpoint_shift_indices + 1 == i) ~= 0
    for q = 1:length(ECU)
        if strcmp(ECU(q).Action, 'Control')
            sp = sp + 1;
            setpoint = setpoints(sp);
            ctrl_ECU(ECU(q).Handles, 'set temperature setpoint', setpoint);
        end
    end
end

%Pause if the allotted timestep has not already elapsed
if seconds_per_datapoint > stoppage_time
    disp('pausing');
pause(seconds_per_datapoint - stoppage_time);

disp('proceeding');

end

end

%% Return ECUs to BCIL mode
for i = 1:length(controlled_ECUs)
    ctrl_ECU(ECU(controlled_ECUs(i))).Handles, 'vent', 1);
end

pause(120);

for i = 1:length(controlled_ECUs)
    ctrl_ECU(ECU(controlled_ECUs(i))).Handles, 'off', 1);
    ctrl_ECU(ECU(controlled ECUs(i))).Handles, 'BCIL mode'
end

%% Sever modbus connection to ECUs
for i = 1:length(ECU)
    if ~strcmp(ECU(i).Action, 'Bypass')
        tcpip_cleanup({ECU(i).Handles})
    end
end

%% Save test data

%Save to workspace
str = ['Test_Data_', Date, '.mat'];
data_array = [format_descriptor; data_array];
save(str, 'data_array');

%Save to spreadsheet
homepath = pwd;
% directory to save
homepath = homepath + ['/Tests/', Date, '/MIT Log/Controller Data'];
filename = homepath(1:end-1);
xlswrite(filename, data_array); % Write data array to Excel spreadsheet

catch
  disp('Caught an Error!');
  for i = 1:length(controlled_ECUs)
    ctrl_ECU(ECU(controlled_ECUs(i))).Handles, 'vent', 1);
  end
  pause(60);

  for i = 1:length(controlled_ECUs)
    ctrl_ECU(ECU(controlled_ECUs(i))).Handles, 'off', 1);
    ctrl_ECU(ECU(controlled_ECUs(i))).Handles, 'BCIL mode'
  end

  disp('Return ECUs to BCIL mode');
  for i = 1:length(ECU)
    if strcmp(ECU(i).Action, 'Bypass')
      tcpip_cleanup({ECU(i).Handles})
    end
  end
end

202
% Save test data

%Save to workspace
str = ['Test_Data_', Date, '.mat'];
data_array = [format_descriptor; data_array];
save(str, 'data_array');

%Save to spreadsheet
homepath = pwd;
cd ../..../..
%  cd(['Tests/', Date, '/MIT Log/Controller Data']);
%  xlswrite(filename, data_array);
%  cd(homepath);

disp('Caught and Error!');
end

B.2.2  ctrl_ECU.m

function result = ctrl_ECU(handles, cmd, arg)
switch cmd
  case {'off', 'vent', 'cool', 'heat'}
    result = change_mode(handles, cmd);
    return
  case 'check mode'
    result = send_command([1, 0, 6, 1, 3, 2003, 1], handles);
    switch result

203
case 0
    result = 'off';
    return

    case 1
    result = 'vent';
    return

    case 2
    result = 'cool';
    return

    case 3
    result = 'heat';
    return
end

    case 'check setpoint'
    result = send_command([1, 0, 6, 1, 3, 2007, 1], handles);
    return

    case 'return temp'
    result = send_command([1, 0, 6, 1, 3, 2010, 1], handles) / 10;
    return

    case 'return humidity'
    result = send_command([1, 0, 6, 1, 3, 2013, 1], handles) / 10;
    return

    case 'output temp'
    result = send_command([1, 0, 6, 1, 3, 2011, 1], handles) / 10;
    return
case 'output humidity'
    result = send_command([1,0,6,1,3,2014,1], handles) / 10;
    return

case 'set temperature setpoint' %degrees fahrenheit (INTEGER)
    if arg != -1
        result = send_command([1,0,6,1,6,4001, arg], handles);
        return
    end

case 'set sample period' %Units of seconds (INTEGER)
    if arg != -1
        result = send_command([1,0,6,1,6,4002, arg], handles);
        return
    end

case 'set integral gain'
    result = send_command([1,0,6,1,6,4003, arg*1000], handles);
    return

case 'set throttling range'
    result = send_command([1,0,6,1,6,4004, arg*10], handles);
    return

case 'set compensation mode'

205
if strcmpi(arg, 'p')
    result = send_command([1,0,6,1,6,4005,0], handles);
elseif strcmpi(arg, 'i')
    result = send_command([1,0,6,1,6,4005,1], handles);
elseif strcmpi(arg, 'pi')
    result = send_command([1,0,6,1,6,4005,2], handles);
end
return

case 'set operating mode'
    if strcmpi(arg, 'vent')
        result = send_command([1,0,6,1,6,4006,1], handles);
    elseif strcmpi(arg, 'cool')
        result = send_command([1,0,6,1,6,4006,2], handles);
    elseif strcmpi(arg, 'heat')
        result = send_command([1,0,6,1,6,4006,3], handles);
    end
    return

case 'query duty cycle'
    result = send_command([1,0,6,1,3,2012,1], handles);
    return

case 'BCIL mode'

result = send_command([1,0,6,1,6,3002,0], handles);
return

case 'MIT mode'
    result = send_command([1,0,6,1,6,3002,1], handles);
    return

case 'override compressor'
    result = send_command([1,0,6,1,6,3003,1], handles);
    return

case 'return compressor control'
    result = send_command([1,0,6,1,6,3003,0], handles);
    return

case 'unload compressor'
    result = send_command([1,0,6,1,6,3004,0], handles);
    return

case 'load compressor'
    result = send_command([1,0,6,1,6,3004,1], handles);
    return

case 'check heater'
    result = send_command([1,0,6,1,3,2015,1], handles);
    return

case 'check compressor'
    result = send_command([1,0,6,1,3,2016,1], handles);
    if result == 1
result = 0;
elseif result == 0
    result = 1;
end
return

case 'check fanspeed'
    result = send_command([1,0,6,1,3,2017,1], handles);
    return

case 'check scroll'
    result = send_command([1,0,6,1,3,2018,1], handles);
    return

% NOT CURRENTLY SUPPORTED IN SOFTWARE
%    case 'heat override'
%        result = ctrl_device_2([1,0,6,1,6,3001,6], handles);
%        return
%    end

result = 'error!';
return

change_mode.m

change_mode.m is an auxiliary function of ctrl_ECU.m, which imposes a delay on
the process that places the ECU into Cool mode. The purpose of this is to operate
the compressor like the ECU’s native compressor controller does.

function result = change_mode(handles, cmd)
```matlab
current_mode = ctrl_ECU(handles, 'check mode');

if strcmp(current_mode, cmd)
    result = cmd;
elseif strcmp(cmd, 'off')
    result = send_command([1,0,6,1,6,3001,0], handles);
elseif strcmp(cmd, 'vent')
    result = send_command([1,0,6,1,6,3001,1], handles);
elseif strcmp(cmd, 'cool')
    result = send_command([1,0,6,1,6,3001,2], handles);
    pause(4)
    ctrl_ECU(handles, 'override compressor', 1);
elseif strcmp(cmd, 'heat')
    result = send_command([1,0,6,1,6,3001,3], handles);
end

B.2.3 send_command

function result = send_command(stor, handles)% action, register, data
% ipaddr    => ip address of the device as a string
% port      => ip address port (Modbus = reserved for 502)
% action    => function code
%Function Code for Modbus
%(0x03) Read Holding Registers
%(0x04) Read Input Registers
%(0x06) Write Single Register
%(0x17) Read/Write Multiple Registers
%register   => Register to perform the action to
```

209
%Example(s):
%Relay 1 write (6) a close operation to register 2001
%ctrl_device('192.168.10.30',1023+relaynum,6,2001,1);
%ctrl_device('192.168.10.30',1024,6,2001,1);
%ctrl_device('192.168.10.30',1030,[100,0,6,1,6,2001,1])

%Relay 1 write (6) a open operation to register 2001
%ctrl_device('192.168.10.30',1024,6,2001,2);

%Woodward read (3) single register 509 (parameter 508?) test
%ctrl_device('192.168.10.35',502,3,509,1);
%ctrl_device('192.168.10.35',502,6,509,200);  %write 200 to 509
%ctrl_device('192.168.10.35',502,3,509,1);
%ctrl_device('192.168.10.35',502,6,509,0);  %write 0 to 509
%ctrl_device('192.168.10.35',502,3,509,1);

%Woodward read (3) and write (6) single register 504 (parameter 503) start
%ctrl_device('192.168.10.35',502,3,504,1);
%ctrl_device('192.168.10.35',502,6,504,1);  %write 1 to bit 0 of 504
%ctrl_device('192.168.10.35',502,3,504,1);
%ctrl_device('192.168.10.35',502,6,504,3);  %write 1 to bit 1 of 504
%ctrl_device('192.168.10.35',502,3,504,1);
%Woodward read (17) multiple registers 5521 (parameter 5520?)
change load
%ctrl_device('192.168.10.35',502,3,5521,2); %,17,5521,2)
%ctrl_device('192.168.10.35',502,17,5521,[0 3000]); %write
300kw to 5521

%Things to improve
%ip address, port, action, register, value
%fault to error or better understand it
%Lengf and
%display of the register and the value
%use unit id as an input?

%Setup the connection first and then reference that later
%Do not open and close connections
%

% Perform the Action
%Once connected then assemble packet and send information
if(handles.connected)
    switch stor(5)
        case {3,4,6} %Read(3) & Write(6) Holding Register;
                      Read(4) Input Reg
                      transID=dec2hex(stor(1),4); %hex transaction
                      identifier
211
ProtID =dec2hex(stor (2) ,4); %hex Protocol ID (0 for ModBus)
Lenghf =dec2hex(stor (3) ,4); %hex Remaining bytes in message
UnitID =dec2hex(stor (4) ,2); %hex Unit ID (usually 1)
FunCod =dec2hex(stor (5) ,2); %hex Function code
Add =dec2hex(stor (6) ,4); %hex Address of the register hi and lo
Val =dec2hex(stor (7) ,4); %hex Data %used for function code 3 and 6
mes=[transID, ProtID, Lenghf, UnitID, FunCod, Add, Val];
% disp(mes);
clearvars translate;
for i=1:length(mes)/2
   translate(i,:)=mes(2*i-1:2*i);
end
% disp(translate);
send=uint8(hex2dec(translate));
ModBusTCP=handles.tcp;
%fwrite(ModBusTCP, send,'uint8 ')
[bytes Data]=writeModBus(handles.tcp, send);
% fprintf('Write Register: %g \t Data: %g \n',
hex2dec(Add),Data)
case{16} %Write Multiple (16) Holding Registers
transID=dec2hex(stor (1) ,4); %hex transaction identifier
ProtID =dec2hex(stor (2) ,4); %hex Protocol ID (0 for ModBus)
Lenghf = dec2hex(stor(3),4); %hex Remaining bytes in message
UnitID = dec2hex(stor(4),2); %hex Unit ID (usually 1)
FunCod = dec2hex(stor(5),2); %hex Function code
Add = dec2hex(stor(6),4); %hex Address of the register hi and lo
Val = dec2hex(stor(7),4); %hex Number of register(s)
ByteCount = dec2hex(stor(8),2); %hex Byte Count left in the message
Dat = dec2hex(stor(9:10),4); %needed for function code 16
mes=[transID, ProtID, Lenghf, UnitID, FunCod, Add, Val, ByteCount, Dat(1,:),Dat(2,:)];
%disp(mes); messaged=hex2dec(mes');
clearvars translate1;
for i=1:length(mes)/2
    translate1(i,:) = mes(2*i-1:2*i);
end
%disp(translate1);
send1=uint8(hex2dec(translate1));
ModBusTCP=handles.tcp;
%fwrite(ModBusTCP, send1,'uint8 ')
[bytes Data]=writeModBus(handles.tcp, send1);
otherwise
    disp('Function not supported at this time')
end
%fclose (handles.tcp);
result=Data; %information collected is passed back out
else
disp(['Error: modbus not connected']);
result=0;
return;
end

B.2.4 format_array.m

function [data_array, format_descriptor] = format_array(ECU, num_rows)
%%% This function formats an array (data_array) to hold the data for an ECU control
%%% test. It also creates a key for reading the resulting data file
%%% (format_descriptor).

data_fields = {'time', ...
   'return air temp',...
   'return air humidity',...
   'duty cycle',...
   'operating mode',...
   'heater state',...
   'compressor state',...
   'scroll state',...
   'setpoint'};

field_members = cell(size(data_fields));
for i = 1:length(data_fields)
    if strcmp(data_fields{i}, 'time')
        field_members{i} = 0;
    elseif strcmp(data_fields{i}, 'return air temp')
        for q = 1:length(ECU)
                field_members{i} = [field_members{i}, ECU(q).Tent];
        end
    end
    elseif strcmp(data_fields{i}, 'return air humidity')
        for q = 1:length(ECU)
                field_members{i} = [field_members{i}, ECU(q).Tent];
        end
    end
    elseif strcmp(data_fields{i}, 'output air temp')
        for q = 1:length(ECU)
                field_members{i} = [field_members{i}, ECU(q).Tent];
        end
    end
elseif strcmp(data_fields{i}, 'output air humidity')
    for q = 1:length(ECU)
            field_members{i} = [field_members{i}, ECU(q).Tent];
        end
    end
end

elseif strcmp(data_fields{i}, 'duty cycle')
    for q = 1:length(ECU)
        if strcmp(ECU(q).Action, 'Control')
            field_members{i} = [field_members{i}, ECU(q).Tent];
        end
    end
end

elseif strcmp(data_fields{i}, 'operating mode')
    for q = 1:length(ECU)
            field_members{i} = [field_members{i}, ECU(q).Tent];
        end
    end
end

elseif strcmp(data_fields{i}, 'heater state')
    for q = 1:length(ECU)
    field_members{i} = [field_members{i}, ECU(q).Tent];
end

elseif strcmp(data_fields{i}, 'compressor state')
    for q = 1:length(ECU)
        if strcmp(ECU(q).Action, 'Monitor') && (strcmp(ECU(q).Compressor_Monitor, 'Y'))
            field_members{i} = [field_members{i}, ECU(q).Tent];
        end
    end
end

elseif strcmp(data_fields{i}, 'scroll state')
    for q = 1:length(ECU)
        if strcmp(ECU(q).Action, 'Control') || (strcmp(ECU(q).Action, 'Monitor') && (strcmp(ECU(q).Compressor_Monitor, 'Y')))
            field_members{i} = [field_members{i}, ECU(q).Tent];
        end
    end
end

elseif strcmp(data_fields{i}, 'setpoint')
    field_members{i} = 0;
end
end

num_columns = 0;
for i = 1:length(field_members)
    num_columns = num_columns + length(field_members{i});
end

colnum = 1;
format_descriptor = cell(1, num_columns);
tent_number = zeros(1, num_columns);

for i = 1:length(field_members)
    format_descriptor(colnum:colnum+length(field_members{i})-1) = {data_fields{i}};
    tent_number(colnum:colnum+length(field_members{i})-1) = field_members{i};
    colnum = colnum+length(field_members{i});
end

for i = 1:num_columns
    if tent_number(i) ~= 0
        format_descriptor{i} = [format_descriptor{i}, ', ', num2str(tent_number(i))];
    end
end

data_array = cell(num_rows, num_columns);

B.2.5 collect_data.m

function datarow = collect_data(ECU, format_descriptor, setpoint, duty)
datarow = cell(1,length(format_descriptor));

for i = 1:length(format_descriptor)
    disp(['Tent ', format_descriptor{i}(end)]);
    if strcmp(format_descriptor{i},'time')
        disp(1);
        datarow{i} = datestr(now);
    elseif strcmp(format_descriptor{i}(1:end-3),'return air temp')
        disp(2);
        datarow{i} = ctrl_ECU(ECU(str2double(format_descriptor{i}(end))).Handles,'return temp',1);
    elseif strcmp(format_descriptor{i}(1:end-3),'return air humidity')
        disp(3);
        datarow{i} = ctrl_ECU(ECU(str2double(format_descriptor{i}(end))).Handles,'return humidity',1);
    elseif strcmp(format_descriptor{i}(1:end-3),'output air temp')
        disp(4);
        datarow{i} = 0;\%ctrl_ECU(ECU(str2double(format_descriptor{i}(end))).Handles,'output temp',1);
elseif strcmp(format_descriptor{i}(1:end-3), 'output air humidity')
    disp(5);
    datarow{i} = 0; %ctrl_ECU(ECU(str2double(
        format_descriptor{i}(end))).Handles, 'output humidity',1);

elseif strcmp(format_descriptor{i}(1:end-3), 'duty cycle')
    disp(6);
    datarow{i} = duty(1, str2double(format_descriptor{i}(end)));

elseif strcmp(format_descriptor{i}(1:end-3), 'operating mode')
    disp(7);
    datarow{i} = ctrl_ECU(ECU(str2double(
        format_descriptor{i}(end))).Handles, 'check mode',1);

elseif strcmp(format_descriptor{i}(1:end-3), 'heater state')
    disp(8);
    datarow{i} = ctrl_ECU(ECU(str2double(
        format_descriptor{i}(end))).Handles, 'check heater',1);

elseif strcmp(format_descriptor{i}(1:end-3), 'compressor state')
disp(9);
datarow{i} = ctrl_ECU(ECU(str2double(
    format_descriptor{i}(end))).Handles, 'check compressor', 1);

elseif strcmp(format_descriptor{i}(1:end-3), 'scroll state')
disp(10);
datarow{i} = ctrl_ECU(ECU(str2double(
    format_descriptor{i}(end))).Handles, 'check scroll', 1);

elseif strcmp(format_descriptor{i}, 'setpoint')
disp(11);
datarow{i} = setpoint;
end
end

B.2.6 scheduler_func.m

function [run_sched] = scheduler_func(duty_cycles,
    num_timesteps, prev_end)

for i = 1:length(duty_cycles)
    if duty_cycles(i) < 0
        duty_cycles(i) = 0;
    end
end

%Sort the ECUs in longest to shortest time
% [duty_cycles, duty_ind] = sort(duty_cycles, 'descend');

% Unsort matrix with correct index
% [~, duty_ind] = sort(duty_ind);

% Minimum # of ECUs that will NEED to run simultaneously in the next period
min_overlap = ceil(sum(duty_cycles)/100);

% Convert requested duty cycles to requested # of timsteps ON for each ECU
steps_req = round(num_timesteps*duty_cycles /100);

% Construct empty run-time schedule to be filled
run_sched = zeros(num_timesteps, length(steps_req));

fin = 0;

% Minimum overlap method
for i = 1:length(steps_req)
    start = fin+1;
    if start+steps_req(i)-1 <= num_timesteps
        % Fill it out!
        run_sched(start:start+steps_req(i)-1, i) = ones(steps_req(i),1);
        fin = start+steps_req(i)-1;
    else
        % Fill it out!
        run_sched(start:start+len(steps_req(i)), i) = ones(len(steps_req(i)),1);
        fin = start+len(steps_req(i));
    end
end
% Fill out to the end, then loop back to the beginning!

run_sched(start:num_timesteps, i)
    = ones(num_timesteps - start + 1, 1);
run_sched(1:steps_req(i) - num_timesteps + start - 1, i)
    = ones(steps_req(i) - num_timesteps + start - 1, 1);
fin = steps_req(i) - num_timesteps + start - 1;
end
end
rnk = zeros(length(run_sched(:,1)),1);

% Circle-shift schedule matrix to match current start state
% to previous end state.
for i = 1:length(run_sched(:,1))
    for s = 0:length(run_sched(1,:))
        if sum(run_sched(i,:) == prev_end) == s
            % Store rank of schedule instance (how many device
            % states match
            % last state of previous schedule)
            rnk(i) = s;
        end
    end
end

% As a first experiment, shift the schedule so that the first instance with
% the highest rank is brought to the front.
\texttt{mx = find(rnk == max(rnk), 1);}
\texttt{run_sched = circshift(run_sched, [-mx+1, 0]);}

\section*{B.3 MATLAB temperature data analysis/display}

The MATLAB temperature data analysis/display suite has 11 major functional components:

- \texttt{compile_data.m}
- \texttt{unify.m}
- \texttt{clip_camp.m}
- \texttt{controller_read.m}
- \texttt{detect_cool_u.m}
- \texttt{detect_cool_c.m}
- \texttt{detect_heat_c.m}
- \texttt{detect_heat_u.m}
- \texttt{hobo_read.m}
- \texttt{weather_read.m}
- \texttt{analyze.m}

\subsection*{B.3.1 compile_data.m}

\texttt{compile_data.m} compiles ECU control test data from three disparate sources—embedded Raspberry Pi ECU controllers, HoBo temperature sensors mounted in the tents being tested, and the BCIL’s weather station database—which may in general all have different start/stop times and different sample rates, and integrates them
into a single data structure that is clipped and interpolated such that all data can be analyzed together.

```matlab
function [SC, SC_u] = compile_data(date, op_mode)
%compile_data('MON DD YYYY') returns two structs:
% 1. SouthCamp: Contains original datapoints from all available sensors
% 2. SouthCamp_u: Contains clipped and interpolated datapoints, such that
% all datapoints are aligned to the same time vector.

% Filename assignment
test_date = date;

h = pwd;
cd ..;/../.../...;
filename_dataguide = [pwd, '\Tests\', test_date, '\MIT Log\DataGuide\', test_date]; % Hobo data guide
filename_controller = [pwd, '\Tests\', test_date, '\MIT Log\Controller Data\ECU\', test_date]; % Controller data
filename_hobo_base = [pwd, '\Tests\', test_date, '\MIT Log\HoBo Data\']; % Hobo sensor data
filename_BCIL_weather = [pwd, '\Tests\', test_date, '\BCIL Log\weather_data.csv']; % BCIL weather data
filename_NILM_IVbase = [];

% NILM IV stream data
filename_NILM_SFbase = [];

% NILM sinefit data
```
data
filename_NILM_PKbase = []; % pwd, '\Tests\', test_date, '\MIT Log\NILM Data\' ]; % NILM sinefit data % NILM sinefit data
data

cd(['Tests', test_date, '\MIT Log']);

 [~, guidetxt, ~] = xlsread(filename_dataguide, 'South Camp', 'B2:14');
guidetxt_ch = cell(size(guidetxt));
num_channels = cell(size(guidetxt));
filename_hobo_output = cell(1,8);
filename_nilm_iv = cell(1,8);
filename_nilm_sf = cell(1,8);
filename_nilm_pk = cell(1,8);

for i = 1:8
    if guidetxt{1,i} == 'N'
        guidetxt_ch{1,i} = guidetxt{1,i}(end-2);
        num_channels{1,i} = guidetxt{1,i}(end);
        guidetxt{1,i} = guidetxt{1,i}(1:find(guidetxt{1,i} == '_')-1);
        filename_hobo_output{1,i} = [filename_hobo_base, guidetxt{1,i}];
    else
        filename_hobo_output{1,i} = 'N';
    end
end

filename_hobo_return = cell(1,8);
for i = 1:8
    if guidetxt{2,i} ~= 'N'
        guidetxt_ch{2,i} = guidetxt{2,i}(end-2);
        num_channels{2,i} = guidetxt{2,i}(end);
        guidetxt{2,i} = guidetxt{2,i}(1:find(guidetxt{2,i} == '_')-1);
        filename_hobo_return{1,i} = [filename_hobo_base,
                                   guidetxt{2,i}];
    else
        filename_hobo_return{1,i} = 'N';
    end
end
for i = 1:8
    if guidetxt{3,i} == 'Y'
        filename_nilm_iv{1,i} = ['NILMiv_T',num2str(i),'.txt'];
        filename_nilm_sf{1,i} = ['NILMsf_T',num2str(i),'.txt'];
        filename_nilm_pk{1,i} = ['NILMpk_T',num2str(i),'.txt'];
    end
end
cd(h);

%% Initialize SouthCamp data structure
data_struct = struct('time',[],'data',[]);
data_struct_multi = struct('controller', data_struct, 'hobo', data_struct);
datastructmulti2 = struct ('BCIL', datastruct);
datastructmulti3 = struct ('NILM', datastruct);

weather_data = struct ('temp', datastructmulti2, ...
  'irradiance', datastructmulti2, ...
  'wind', datastructmulti2, ...
  'humidity', datastructmulti2);

tentstruct = struct ('return_temp', datastructmulti, ...
  'return_humidity', datastructmulti, ...
  'output_temp', datastructmulti, ...
  'output_humidity', datastructmulti, ...
  'mode', datastructmulti, ...
  'solenoid', datastructmulti, ...
  'duty', datastructmulti, ...
  'op_mode', datastructmulti, ...
  'heater_state', datastructmulti, ...
  'compressor_state', datastructmulti, ...
  'scroll_state', datastructmulti, ...
  'setpoint', datastructmulti, ...
  'V_a', datastructmulti3, ...
  'V_b', datastructmulti3, ...
  'V_c', datastructmulti3, ...
  'SF', datastructmulti3, ...
  'PK', datastructmulti3, ...
  'I_a', datastructmulti3, ...
  'I_b', datastructmulti3, ...
  'I_c', datastructmulti3);

Tent = [tentstruct ... Tent]
tent struct... %Tent 2
tent struct... %Tent 3
tent struct... %Tent 4
tent struct... %Tent 5
tent struct... %Tent 6
tent struct... %Tent 7
tent struct... %Tent 8

tentstruct...

SouthCamp = struct ('Tent', Tent, 'Weather', weather_data);

% read data from specified locations
if ~isempty(filename_controller)
    try
        SouthCamp = controller_read(SouthCamp, filename_controller);
    catch
        disp(['Error occurred when attempting to read controller data from ', filename_controller, '!'])
    end
end

for i = 1:8
    try
        if ~isempty(filename_hobo_return{1,i})
            SouthCamp.Tent(i).return_temp.hobo = hobo_read(
                SouthCamp.Tent(i).return_temp.hobo,
                filename_hobo_return{1,i}, guidetxt_ch{2,i},
            )
        end
    catch
        disp(['Error occurred when attempting to read hobo data from ', filename_hobo_return{1,i}, '!'])
    end
end
num_channels{2,i});

  end

  catch
    disp([^Error occurred when attempting to read hobo
         data from ', filename_hobo_return{1,i}, '!']);
  end

  try
    if ~isempty(filename_hobo_output{1,i})
      SouthCamp.Tent(i).output_temp.hobo = hobo_read(
        SouthCamp.Tent(i).output_temp.hobo,
        filename_hobo_output{1,i}, guidetxt_ch{1,i},
        num_channels{1,i});
    end
  catch
    disp([^Error occurred when attempting to read hobo
         data from ', filename_hobo_output{1,i}, '!']);
  end

  end

  if ~isempty(filename_BCIL_weather)
    try
      SouthCamp.Weather = weather_read(SouthCamp.Weather,
                                         filename_BCIL_weather);
    catch
      disp([^Error occurred when attempting to read weather
           data from ', filename_BCIL_weather, '!']);
    end
  end

  % SouthCamp = NILMiv_read(SouthCamp, filename_NILM_IVbase,
filename_nilm_iv);

% SouthCamp = NILMsf_read(SouthCamp, filename_NILM_SFbase,
    filename_nilm_sf);
SouthCamp = NILMpk_read(SouthCamp, filename_NILM_PKbase,
    filename_nilm_pk);

if strcmpi(op_mode, 'cool')
    for i = 1:8
        if ~isempty(SouthCamp.Tent(i).compressor_state.
            controller.data)
            SouthCamp.Tent(i).compressor_state.controller.
                data(SouthCamp.Tent(i).compressor_state.
                    controller.data == 65535) = 0;
        end
    end
end

SC_u = crisp_modedata(unify(SouthCamp), op_mode);
SC = SouthCamp;

% package_iddata(SC_u, -1, -1, test_date);
save(['C:\Users\2016\Dropbox (MIT)\BCIL\Code\MATLAB\BCIL Test
        Data Analysis\Compiled Data\', test_date, ' Post-
        Processing.mat'], 'SC*');

B.3.2 unify.m

unify.m is responsible for matching the start/stop times and sample rates of all data
vectors compiled by compile_data.m. It does this by choosing the latest start and
the earliest end time as the start and end times for all datasets and clipping them
accordingly, and then interpolating all datasets according to the time vector of the
data sequence with the highest sample rate.

```matlab
function SouthCamp_new = unify(SouthCamp)

earliest_end = inf;
latest_start = -inf;

field0 = fieldnames(SouthCamp);
for h = 1:length(field0)
    for i = 1:length(SouthCamp.(field0{h}))
        field1 = fieldnames(SouthCamp.(field0{h})(i));
        for q = 1:length(field1)
            disp(['field0{h}, ', 'field1{q}']);
            field2 = fieldnames(SouthCamp.(field0{h})(i).(field1{q}));
            for r = 1:length(field2)
                if ~isempty(SouthCamp.(field0{h})(i).(field1{q}).(field2{r}).time)
                    if SouthCamp.(field0{h})(i).(field1{q}).(field2{r}).time(end) <= earliest_end
                        earliest_end = SouthCamp.(field0{h})(i).(field1{q}).(field2{r}).time(end);
                    end
                end
                if SouthCamp.(field0{h})(i).(field1{q}).(field2{r}).time(1) >= latest_start
                    latest_start = SouthCamp.(field0{h})(i).(field1{q}).(field2{r}).time(1);
                end
            end
        end
    end
end
```
res = inf;
highest_res = -1;

for h = 1:length(field0)
    for i = 1:length(SouthCamp.(field0{h}))(i))
        field1 = fieldnames(SouthCamp.(field0{h})).(i).
        for q = 1:length(field1)
            disp([field0{h}, ' ', field1{q}]);
            field2 = fieldnames(SouthCamp.(field0{h}))(i).
                for r = 1:length(field2)
                    if ~isempty(SouthCamp.(field0{h}))(i).
                        field1{q}),(field2{r}).time)
                            %Interpolate data vector to match highest
                            %resolution, shift
                            %to unified time vector
                            if mean(diff(SouthCamp.(field0{h}))(i).
                                field1{q}),(field2{r}).time)) <= res
                                res = mean(diff(SouthCamp.(field0{h})
                                    (i).(field1{q}),(field2{r}).time))
                            ;
                            highest_res = SouthCamp.(field0{h}))(i
                                .(field1{q}),(field2{r}).time);
end
end
end

field0 = fieldnames(SouthCamp);
for h = 1:length(field0)
    for i = 1:length(SouthCamp.(field0{h}))
        field1 = fieldnames(SouthCamp.(field0{h})(i));
        for q = 1:length(field1)
            disp([field0{h}, ' ', field1{q}]);
            field2 = fieldnames(SouthCamp.(field0{h})(i).(
                field1{q}));
            for r = 1:length(field2)
                if ~isempty(SouthCamp.(field0{h})(i).(field1{
                    q}).(field2{r}).time) && ~strcmp(field1{q}, 'op_mode')
                    % Interpolate data vector to match highest resolution, shift
                    % to unified time vector
                    disp([field0{h}, ' ', field1{q}, ' ', field2{r}]);
                    SouthCamp.(field0{h})(i).(field1{q}).(field2{r}).data = interp1(SouthCamp.(field0{h})(i).(field1{q}).(field2{r}).time, double(SouthCamp.(field0{h})(i).(field1{q}).(field2{r}).data),
                        highest_res);
                end
            end
        end
    end
end
field2{r}).time = highest_res;

end

end
end
end
end
end

field0 = fieldnames(SouthCamp);

for h = 1:length(field0)
    for i = 1:length(SouthCamp.(field0{h})){
        field1 = fieldnames(SouthCamp.(field0{h})){i}.
        field2 = fieldnames(SouthCamp.(field0{h})){i}.
        field2{r} = fieldnames(SouthCamp.(field0{h})){i}.

    for r = 1:length(field2)
        %Clip all non-empty time vectors to uniform start/end
        if ~isempty(SouthCamp.(field0{h})){i}.
            %Clip beginning and end times to match
            [-, start] = min(abs(SouthCamp.(field0{h})){i}.
                field1{q}.
                field2{r}.
            [-, fin] = min(abs(SouthCamp.(field0{h})){i}.
                field1{q}.

        SouthCamp.(field0{h})){i}.
            field2{r}.
        end
    end
end

83 82

81

80

79

78

77

76

75

74

73

72

71

70

69

68

67

66

65

64

63

62

61

60

59

58

57

56

55

54

53

52

51

50

49

48

47

46

45

44

43

42

41

40

39

38

37

36

35

34

33

32

31

30

29

28

27

26

25

24

23

22

21

20

19

18

17

16

15

14

13

12

11

10

9

8

7

6

5

4

3

2

1

0
B.3.3 clip_camp.m

clip_camp.m trims data off of the ends of data sequences as prescribed by unify.m.

function [camp_new, ind_st, ind_nd] = clip_camp(camp_old, t_start, t_end)

%% This function only works with the TIME-UNIFIED instance of the camp
%% structure!

%Find start and end indices
if ~isempty(camp_old.Tent(1).return_temp.controller.time)
    ind_st = find(camp_old.Tent(1).return_temp.controller.time == datenum(t_start));
end

ind_st = find(camp_old.Tent(5).return_temp.controller.time == datenum(t_end));
time = datenum(t_start);

ind_nd = find(camp_old.Tent(5).return_temp.controller.

time = datenum(t_end));

end

field0 = fieldnames(camp_old);

for h = 1:length(field0)
    for i = 1:length(camp_old.(field0{h})){i});
        field1 = fieldnames(camp_old.(field0{h})){i}.
        field1{q});
    for r = 1:length(field2)
        %Clip all non-empty time vectors to
        uniform start/end
        if ~isempty(camp_old.(field0{h})){i}.
            field1{q}.
            field2{r}).time) && ~strcmpi(field1{q}.
            'op_mode'
        %Interpolate data vector to match highest
        resolution, shift
        %to unified time vector
        disp([field0{h},'.',field1{q},'.',field2{r}]);
        camp_old.(field0{h}){i}.
            field1{q}.
            field2{r}).data = camp_old.(field0{h})
            (i).field1{q}.
            field2{r}).data(
            ind_st:ind_nd);
        camp_old.(field0{h}){i}.
            field1{q}.
            field2{r}).time = camp_old.(field0{h})
            (i).field1{q}.
            field2{r}).time(
B.3.4 controller_read.m

controller_read.m reads all data gathered from embedded Raspberry Pi ECU controllers during testing.

```matlab
function SC = controller_read(SC, filename)

read_mode = 0; % to accomodate dysfunctional heater/compressor recordings, 0 for normal datalogs!

time_mode = 0; % to accomodate dysfunctional time recordings (non-fractional seconds), 0 for normal datalogs!

[~,raw] = xlsread(filename);
num_series = length(raw(1,:));

for i = 1:num_series
    if strcmp(raw(1,i), 'time')
        time = datenum(raw(2:end,i));
        if time_mode == 1
            time = linspace(time(1), time(end), length(time));
        end
    end
```
elseif strcmp(raw{1, i}(1:end-3), 'return air temp')
    data = [raw{2:end, i}]';
    n = str2num(raw{1, i}(end));
    SC.Tent(n).return_temp.controller.data = data;
    SC.Tent(n).return_temp.controller.time = time;

elseif strcmp(raw{1, i}(1:end-3), 'return air humidity')
    data = [raw{2:end, i}]';
    n = str2num(raw{1, i}(end));
    SC.Tent(n).return_humidity.controller.data = data;
    SC.Tent(n).return_humidity.controller.time = time;

elseif strcmp(raw{1, i}(1:end-3), 'output air temp')
    data = [raw{2:end, i}]';
    n = str2num(raw{1, i}(end));
    SC.Tent(n).output_temp.controller.data = data;
    SC.Tent(n).output_temp.controller.time = time;

elseif strcmp(raw{1, i}(1:end-3), 'output air humidity')
    data = [raw{2:end, i}]';
    n = str2num(raw{1, i}(end));
    SC.Tent(n).output_humidity.controller.data = data;
    SC.Tent(n).output_humidity.controller.time = time;

elseif strcmp(raw{1, i}(1:end-3), 'duty cycle')
    data = [raw{2:end, i}]';
    n = str2num(raw{1, i}(end));
    SC.Tent(n).duty.controller.data = data;
    SC.Tent(n).duty.controller.time = time;
elseif strcmp(raw{1, i}(1:end-3), 'operating mode')
    data = [raw(2:end, i)]';
    n = str2num(raw{1, i}{end});
    SC.Tent(n).op_mode.controller.data = data;
    SC.Tent(n).op_mode.controller.time = time;
    SC.Tent(n).compressor_state.controller.data = strcmp(
        SC.Tent(n).op_mode.controller.data, 'cool');
    SC.Tent(n).compressor_state.controller.time = time;

elseif strcmp(raw{1, i}(1:end-3), 'heater state')
    if readmode == 0
        data = [raw(2:end, i)]';
        n = str2num(raw{1, i}{end});
        SC.Tent(n).heater_state.controller.data = data;
        SC.Tent(n).heater_state.controller.time = time;
    else
        data = strcmp(SC.Tent(n).op_mode.controller.data,
        ['heat'] | strcmp(SC.Tent(n).op_mode.
        controller.data, ['cool']);
        n = str2num(raw{1, i}{end});
        SC.Tent(n).heater_state.controller.data = data;
        SC.Tent(n).heater_state.controller.time = time;
    end

elseif strcmp(raw{1, i}(1:end-3), 'vent state')
    if read_mode == 0
        ;
    else
        ;
end

elseif strcmp(raw{1, i}(1:end-3), 'compressor state')
data = [raw{2:end, i}]';
n = str2num(raw{1, i}(end));
SC.Tent(n).compressor_state.controller.data = data;
SC.Tent(n).compressor_state.controller.time = time;

elseif strcmp(raw{1, i}(1:end-3), 'scroll state')
data = [raw{2:end, i}]';
n = str2num(raw{1, i}(end));
SC.Tent(n).scroll_state.controller.data = data;
SC.Tent(n).scroll_state.controller.time = time;
end
end

B.3.5 detect_cool_u.m

detect_cool_u.m reads compressor operation data gathered from embedded Raspberry Pi ECU controllers that were configured to just monitor and not control the ECU during testing.

1 function [Tent, ons, offs] = detect_cool_u(Tent, t_start, t_end)

2

3 \% Find the compressor scroll ons and offs

4 \% Only care about what the scroll is doing when the compressor is running.
5 Tent.scroll_state.controller.data = Tent.scroll_state.controller.data & Tent.compressor_state.controller.data;
ons = find((Tent.scroll_state.controller.data == 1) & ([0; diff(Tent.scroll_state.controller.data)] > 0));
ontimes = Tent.scroll_state.controller.time(ons);
for s = 1:length(ontimes)
    if (ontimes(s) < datenum(t_start)) || (ontimes(s) > datenum(t_end))
        ontimes(s) = -1;
    end
end

ons = ontimes(~ = -1);

offs = find((Tent.scroll_state.controller.data == 0) & ([0; diff(Tent.scroll_state.controller.data)] < 0));
offtimes = Tent.scroll_state.controller.time(offs);
for s = 1:length(offtimes)
    if (offtimes(s) < datenum(t_start)) || (offtimes(s) > datenum(t_end))
        offtimes(s) = -1;
    end
end
offs = offtimes(~ = -1);

ons_real = zeros(1,length(ons));
offs_real = zeros(1,length(offs));
for s = 1:length(ons)
    ons_real(s) = find(abs(Tent.compressor_state.controller.time - Tent.compressor_state.controller.time(ons(s))))
for \( s = 1 : \text{length}(\text{offs}) \)
offs_real(s) = find(abs(Tent.compressor_state.controller.time - Tent.compressor_state.controller.time(offs(s))) == min(abs(Tent.compressor_state.controller.time - Tent.compressor_state.controller.time(offs(s)))),1)';
end

ons = ons_real';
offs = offs_real';

\section*{B.3.6 detect_cool_c.m}

detect_cool_c.m reads compressor operation data gathered from embedded Raspberry Pi ECU controllers that were configured to control the ECU during testing.

function [ons, offs] = detect_cool_c(Tent, Tent_orig, t_start, t_end)

\% Here's where code that accounts for the short-cycle timer will go. If the controlled ECUs eventually end up using a direct measurement to report the compressor state, then this section will not be necessary.

\% THIS SECTION IS NOT DONE. IF THE CONTROLLED ECU SWITCHES FROM COOL TO VENT

\% DURING OPERATION, THEN THIS SECTION MUST BE FINISHED.
compressor_offs = find((Tent_orig.compressor_state.controller.data' == 1) & (\mid\text{diff}(Tent_orig.compressor_state.data)')
controller.data')); 0] < 0});

% Find the compressor scroll ons and offs

% Only care about what the scroll is doing when the compressor is running.
Tent_orig.scroll_state.controller.data = Tent_orig.
  scroll_state.controller.data & Tent_orig.compressor_state.
  controller.data';

ons = find((Tent_orig.scroll_state.controller.data == 1) &
  ([0; diff(Tent_orig.scroll_state.controller.data)] > 0));
ontimes = Tent_orig.scroll_state.controller.time(ons);
for s = 1:length(ontimes)
  if (ontimes(s) < datenum(t_start)) || (ontimes(s) >
    datenum(t_end))
    ontimes(s) = -1;
  end
end

ons = ons(ontimes ~= -1);

offs = find((Tent_orig.scroll_state.controller.data == 0) &
  ([0; diff(Tent_orig.scroll_state.controller.data)] < 0));
offtimes = Tent_orig.scroll_state.controller.time(offs);
for s = 1:length(offtimes)
  if (offtimes(s) < datenum(t_start)) || (offtimes(s) >
    datenum(t_end))
    offtimes(s) = -1;
  end
offs = offtimes ~= -1;

ons_real = zeros(1, length(ons));
offs_real = zeros(1, length(offs));

for s = 1:length(ons)
    on_real(s) = find(abs(Tent.compressor_state.controller.time - Tent_orig.compressor_state.controller.time(ons(s))) == min(abs(Tent.compressor_state.controller.time - Tent_orig.compressor_state.controller.time(ons(s)))), 1);
end

for s = 1:length(offs)
    offs_real(s) = find(abs(Tent.compressor_state.controller.time - Tent_orig.compressor_state.controller.time(offs(s))) == min(abs(Tent.compressor_state.controller.time - Tent_orig.compressor_state.controller.time(offs(s)))), 1);
end

ons = on_real';
offs = offs_real';

B.3.7 detect_heat_c.m

detect_heat_c.m reads compressor operation data gathered from embedded Raspberry Pi ECU controllers that were configured to control the ECU during testing.

function [ons, offs] = detect_heat_c(Tent, Tent_orig, t_start, t_end)
ons = find((Tent_orig.mode.controller.data == 2) & ([0; diff(Tent_orig.mode.controller.data)] > 0));
ontimes = Tent_orig.mode.controller.time(ons);
for s = 1:length(ontimes)
    if (ontimes(s) < datenum(t_start)) || (ontimes(s) > datenum(t_end))
        ontimes(s) = -1;
    end
end
ons = ons(find(ontimes ~= -1));
offs = find((Tent_orig.mode.controller.data == 1) & ([0; diff(Tent_orig.mode.controller.data)] < 0));
offtimes = Tent_orig.mode.controller.time(offs);
for s = 1:length(offtimes)
    if (offtimes(s) < datenum(t_start)) || (offtimes(s) > datenum(t_end))
        offtimes(s) = -1;
    end
end
offs = offs(find(offtimes ~= -1));
ons_real = zeros(1,length(ons));
offs_real = zeros(1,length(offs));
for s = 1:length(ons)
    ons_real(s) = find(abs(Tent.mode.controller.time - Tent_orig.mode.controller.time(ons(s))) == min(abs(Tent.mode.controller.time - Tent_orig.mode.controller.time(ons(s))),1),1)';
```matlab
end

for s = 1:length(offs)
    offs_real(s) = find(abs(Tent.mode.controller.time -
                           Tent_orig.mode.controller.time(offs(s))) == min(abs(
                           Tent.mode.controller.time - Tent_orig.mode.controller.
                           time(offs(s))),1))';
end

ons = ons_real';
offs = offs_real';
```

**B.3.8 detect_heat_u.m**

detect_heat_u.m reads compressor operation data gathered from embedded Raspberry Pi ECU controllers that were configured to just monitor and not control the ECU during testing. It identifies heater on/off events by locating minima and maxima in the ECU output air temperature.

```matlab
function [tent_new, ons, offs] = detect_heat_u(Tent)

if ~isempty(Tent.output_temp.hobo.data)
    %Take the first differences of each uncontrolled ECU's OUTPUT air
    %temperature
    diffs = [0; diff(Tent.output_temp.hobo.data)];
    diffs2 = [0; diff(diffs)];

    %For DEBUGGING/CALIBRATING— can comment out
    figure();
    plot(Tent.output_temp.hobo.time, diffs, 'r', Tent.
        output_temp.hobo.time, diffs2, 'k');
```
pre_ons1 = (diffs > 0.25);
pre_ons2 = pre_ons1; % & (diffs2 > 0.25);

pre_offs1 = (diffs < -0.25);
pre_offs2 = pre_offs1; % & (diffs2 < -0.25);

% For DEBUGGING/CALIBRATING— can comment out
figure();
yyaxis right;
plot(Tent.output_temp.hobo.time, Tent.output_temp.hobo.data, 'r');

yyaxis left;
plot(Tent.output_temp.hobo.time, pre_ons2, 'k');
hold on;
plot(Tent.output_temp.hobo.time, pre_offs2, 'p');

on_off = zeros(1, length(Tent.output_temp.hobo.time));
ons = [];
offs = [];

for x = 1:length(on_off)
    if (on_off(x) == 0) & (pre_ons2(x) == 1)
        on_off(x:end) = 1;
        ons = [ons; x];
    elseif (on_off(x) == 1) & (pre_offs2(x) == 1)
        on_off(x:end) = 0;
        offs = [offs; x];
    end
end
Tent.mode.hobo.data = on_off' + ones(size(on_off'));  
Tent.mode.hobo.time = Tent.output_temp.hobo.time;  
tent_new = Tent;  
tent_new.heater_state.hobo.data = zeros(length(Tent.output_temp.hobo.data), 1);  
tent_new.heater_state.hobo.time = tent_new.output_temp.hobo.time;  
for i = 1:length(ons)  
tent_new.heater_state.hobo.data(ons(i):end) = ones(size(tent_new.heater_state.hobo.data(ons(i):end)));  
tent_new.heater_state.hobo.data(offs(i):end) = ones(size(tent_new.heater_state.hobo.data(offs(i):end)));  
end  
if ~isempty(Tent.output_temp.controller.data)  
%Use controller data to detect mode  
end  
return
B.3.9  hobo_read.m

hobo_read.m loads temperature data recorded by HoBo temperature sensors and allows for those data to be integrated by compile_data.py.

```
function hobo_struct_new = hobo_read(hobo_struct, filename, ch, num)

if ~strcmp(filename, 'N')
    [~,~,raw] = xlsread(filename);
    hobo_struct.time = datenum(raw(3:end,2));
    hobo_struct.data = str2double(raw(3:end,str2double(ch)+2));
end

hobo_struct_new = hobo_struct;
```

B.3.10  weather_read.m

weather_read.m loads data from the BCIL's weather station by reading it from an excel sheet.

```
function weather_struct_new = weather_read(weather_struct, filename)

read_mode = 1;

if read_mode == 0
    [num, txt, raw] = xlsread(filename, 'Weather');
    %Read weather data with OLD format
```
weather_struct.temp.BCIL.time = datenum(txt(2:end-1,1));
weather_struct.temp.BCIL.data = num(:,1);
weather_struct.humidity.BCIL.time = weather_struct.temp.BCIL.time;
weather_struct.humidity.BCIL.data = num(:,2);
else
    data = readtable(filename,'HeaderLines',1,'Format','% u %s %f %f %f %f %f %f %f %f %f %f %f %f %f%%f');
%Read weather data with NEW format
    time = datenum(data(:,2){1}(2:end-1));
time = datenum(data(:,2));
weather_struct.pressure.BCIL.time = time;
weather_struct.pressure.BCIL.data = data(:,3);
weather_struct.temp.BCIL.time = time;
weather_struct.temp.BCIL.data = data(:,4);
weather_struct.dewpoint.BCIL.time = time;
weather_struct.dewpoint.BCIL.data = data(:,5);
weather_struct.windchill.BCIL.time = time;
weather_struct.windchill.BCIL.data = data(:,6);
weather_struct.humidity_in.BCIL.time = time;
weather_struct.humidity_in.BCIL.data = data(:,7);
weather_struct.humidity_out.BCIL.time = time;
weather_struct.humidity_out.BCIL.data = data(:,8);
weather_struct.dailyrain.BCIL.time = time;
weather_struct.dailyrain.BCIL.data = data(:,9);
weather_struct.windspeed.BCIL.time = time;
weather_struct.windspeed.BCIL.data = data(:,10);
weather_struct.winddir.BCIL.time = time;
weather_struct.winddir.BCIL.data = data(:,11);
```matlab
weather_struct.rainrate.BCIL.time = time;
weather_struct.rainrate.BCIL.data = data{:,13};
weather_struct.ET.BCIL.time = time;
weather_struct.ET.BCIL.data = data{:,14};
weather_struct.solarrad.BCIL.time = time;
weather_struct.solarrad.BCIL.data = data{:,15};
weather_struct.UV.BCIL.time = time;
weather_struct.UV.BCIL.data = data{:,16};
weather_struct.heatindex.BCIL.time = time;
weather_struct.heatindex.BCIL.data = data{:,17};
end

weather_struct_new = weather_struct;
```

**B.3.11 analyze.m**

analyze.m displays the data from a test after it has been compiled by compile_data.m.

```matlab
clear all;
clf;
close all;

%%% Select Test Parameters
%%% Pick date of test!
% test_date = 'MAR 02 2017'; %MON DD YYYY
test_date = 'SEP 07 2017';
% test_date = 'MAY 22 2017';
ctrl_mode = 'cool';

%%% Pick test start and end time!
t_start = '07-SEP-2017 12:30:00'; %DD-MON-YYYY HH:MM:SS
```
t_end = '07-SEP-2017 14:00:00';

uncontrolled = 5:8;  \%Specify which ECUs are uncontrolled
controlled = 1:4;  \%Specify which ECUs are controlled

\% Load dataset

\% use for Spencer

cd ..
load (pwd, '\Compiled Data\', test_date, ' Post-Processing.mat'])

\% use for pete

\% load ([pwd, '/ Compiled Data/', test_date, ' Post-Processing.mat']

\% cd ('/Users/lindahl/Dropbox (MIT)/BCIL/Code/MATLAB/BCIL Test Data Analysis')
\% load ('/Users/lindahl/Dropbox (MIT)/BCIL/Code/MATLAB/BCIL Test Data Analysis/Compiled Data/MAY 15 2017 Post-Processing.mat');

fprintf(['TENT DATA INDEXING:
', ...'
SC_u.Tent(n).<data_type>.<controller/hobo>.data...
', ...'
', ...'
data_type can be return_temp, return_humidity,
output_temp, output_humidity, mode, or solenoid
', ...]
'not all data fields are populated!\n\n', ...

'WEATHER DATA INDEXING: \n
', ...

'SC_u.Weather.<data_type>.BCIL.data ... \n
', ...

'data_type can be temp, irradiance, wind, or humidity\n
', ...

'not all data fields are populated!\n
'));

8

43

%% Clip Data to Specified Time-Bounds

44

%Make a special instance of SC_u to be manipulated for
plotting

46

SC_p = SC_u;

47 for i = 1:length(controlled)

48 if strcmp(ctrl_mode, 'cool');

49 SC_u.Tent(controlled(i)).scroll_state.controller.data

51 (SC_u.Tent(controlled(i)).scroll_state.controller.

53 data == 65535) = 0;

55 end

end

57

%Clip all datasets of SC_p down to specified start and stop
times

58 [SC_p, ind_start, ind_end] = clip_camp(SC_p, t_start, t_end);

59 SC_p.time = SC_p.Tent(controlled(1)).return_temp.controller.

60 time;

62

%% Independently-Operated HVAC Performance Plots

64

254
%Create empty cells for the on and off times of each HVAC
ons = cell(1,8);
offs = ons;

%Estimate load profile of each uncontrolled tent, and
generate vectors
%containing the indices of each turn-on and turn-off
if strcmpi(ctrl_mode, 'heat')
    for i = 1:length(uncontrolled)
        [SC_u.Tent(uncontrolled(i)), ons{uncontrolled(i)},
         offs{uncontrolled(i)}] = detect_heat_u(SC_p.Tent(uncontrolled(i)));
        SC_p.Tent(uncontrolled(i)) = SC_u.Tent(uncontrolled(i));
    end
elseif strcmpi(ctrl_mode, 'cool')
    for i = 1:length(uncontrolled)
        [SC_u.Tent(uncontrolled(i)), ons{uncontrolled(i)},
         offs{uncontrolled(i)}] = detect_cool_u(SC_p.Tent(uncontrolled(i)),
         SC_p.Tent(uncontrolled(i)), t_start, t_end);
        SC_p.Tent(uncontrolled(i)) = SC_u.Tent(uncontrolled(i));
    end
end

%Construct timeseries representing the total number of
controlled HVACs
%running
tot_run_u = zeros(size(SC_p.Tent(uncontrolled(1))).
            scroll_state.controller.data));
for i = 1:length(uncontrolled)
    if strcmpi(ctrl_mode, 'cool')
        tot_run_u = tot_run_u + SC_p.Tent(uncontrolled(i)).scroll_state.controller.data;
    else
        tot_run_u = tot_run_u + SC_p.Tent(uncontrolled(i)).heater_state.hobo.data;
    end
end

%Add initial turn-on or final turn-off events to make the dataset self-contained.
for i = 1:length(uncontrolled);
    if ~isempty(ons{uncontrolled(i)}) && (ons{uncontrolled(i)}(end) > offs{uncontrolled(i)}(end))
        offs{uncontrolled(i)} = [offs{uncontrolled(i)}; length(SC_p.time)]; %m = length of column vectors in ons and offs arrays
    end
    if offs{uncontrolled(i)}(1)<ons{uncontrolled(i)}(1)
        ons{uncontrolled(i)} = [1; ons{uncontrolled(i)}];
    end
end

%PLOT ONE TOP: Total # ECUs Running
figure(1);
alt = subplot(2,1,1);
stairs(SC_p.time, tot_run_u);
xlabel('Time')
ylabel('# ECUs Running (out of 4)');
title(['Load Profile with Independent HVAC Operation: ', test_date]);
dateaxis;
xlim([SC_p.time(1), SC_p.time(end)]);
ylim([0, 4.2]);

% PLOT ONE BOTTOM: Thermal Performance
alb = subplot(2,1,2);
hold on; color = ['r', 'g', 'b', 'k'];
plb = cell(1,4);
mx = zeros(length(uncontrolled),1);
mn = zeros(length(uncontrolled),1);
for i = 1:length(uncontrolled)
    plb{i} = plot(SC_p.Tent(uncontrolled(i)).scroll_state.
        controller.time, SC_p.Tent(uncontrolled(i)).
        return_temp.hobo.data, color(i));
    mx(i) = max(SC_p.Tent(uncontrolled(i)).return_temp.hobo.
        data);
    mn(i) = min(SC_p.Tent(uncontrolled(i)).return_temp.hobo.
        data);
end
xlabel('Time');
ylabel('Temperature (°F)');
title(['Temperature Performance with Independent HVAC
    Operation: ', test_date]);
dateaxis;
xlim([SC_p.time(1), SC_p.time(end)]);
ylim([min(mn) - 2, max(mx) + 2]);
% Generate indices of each turn-on and turn-off for controlled tents, based on the original dataset. Then, translate these indices to the unified dataset.

if strcmpi(ctrl_mode, 'heat')
    for i = 1:length(controlled)
        [ons{controlled(i)}, offs{controlled(i)}] =
            detect_heat_c(SC_p.Tent(controlled(i)), SC.Tent(controlled(i)), t_start, t_end);
    end
elseif strcmpi(ctrl_mode, 'cool')
    for i = 1:length(controlled)
        [ons{controlled(i)}, offs{controlled(i)}] =
            detect_cool_c(SC_p.Tent(controlled(i)), SC.Tent(controlled(i)), t_start, t_end);
    end
end

% Construct timeseries representing the total number of controlled HVACs
% running

tot_run_c = zeros(length(SC_p.Tent(controlled(1))).

Wo Networked HVAC Performance Plots
scroll_state.controller.data),1);

for i = 1:length(controlled)
    tot_run_c = tot_run_c + SC_p.Tent(controlled(i)).scroll_state.controller.data;
end

%Add initial turn-on or final turn-off events to make the dataset
%self-contained.
for i = 1:length(controlled);
    if ~isempty(ons{controlled(i)}) && (ons{controlled(i)}(end) > offs{controlled(i)}(end))
        offs{controlled(i)} = [offs{controlled(i)}; length(SC_p.time)]; %m = length of column vectors in ons and offs arrays
    end
    if offs{controlled(i)}(1)<ons{controlled(i)}(1)
        ons{controlled(i)} = [1; ons{controlled(i)}];
    end
end

%PLOT TWO TOP: Total # ECUs Running
figure(2);
a2t = subplot(2,1,1);
stairs(SC_p.time, tot_run_c);
xlabel('Time')
ylabel('# ECUs Running (out of 4)');
title(['Load Profile with Networked HVAC Operation: ', test_date]);
datetickzoom;
% PLOT TWO BOTTOM: Thermal Performance

```matlab
axes = subplot(2,1,2);
hold on; color = ['r', 'g', 'b', 'k'];

for i = 1:length(controlled)
    p2b{i} = plot(SC_p.time, SC_p.Tent(controlled(i)).return_temp.hobo.data, color(i));
end

xlabel('Time');
ylabel('Temperature ($^\circ$F)');
title(['Temperature Performance with Networked HVAC Operation :
    ', test_date]);
datetickzoom;

xlim([SC_p.time(1), SC_p.time(end)]);
ylim([min(mn)-2, max(mx)+2]);
legend(['Tent ', num2str(controlled(1))], ['Tent ', num2str(controlled(2))], ['Tent ', num2str(controlled(3))], ['Tent ', num2str(controlled(4))]);
linkaxes([a2t, a2b], 'x');
```

% Independent/Networked HVAC Run-Time Comparison

```matlab
figure(3);
axes = subplot(2,1,1)
plot(SC_p.time,-1*ones(1,length(SC_p.time)));
hold on;
for z = 1:length(uncontrolled)
    for xx = 1:length(ons{uncontrolled(z)})
```
%plots a rectangle with its bottom-left vertex at on
\{z\}, z=0.5; The
%rectangle ends when the next turn-off occurs, and is
0.8 units
% tall.
rectangle('Position', [SC_p.time(ons{uncontrolled(z)}{(xx)), uncontrolled(z)-0.5, SC_p.time(offs{
uncontrolled(z)}(xx)) - SC_p.time(ons{uncontrolled(z)}(xx)), 0.8], 'FaceColor', color(z));

end
end
datetickzoom('x', 'HHMM');
xlim([SC_p.time(1), SC_p.time(end)]);
ylim([uncontrolled(1)-1, uncontrolled(end)+1]);
title('HVAC On-Time Without Interleaving Control');
xlabel('Time (HHMM)', 'FontSize', 12);
ylabel('Tent Number', 'FontSize', 12);
box on
ax = gca;
an.YTickMode = 'manual';
an.YTick = [uncontrolled(1) uncontrolled(2) uncontrolled(3)
uncontrolled(4)];

a3b = subplot(2,1,2);
plot(SC_p.time, -1*ones(1, length(SC_p.time)));
hold on;
for z = 1:length(controlled)
    for xx = 1:length(ons{controlled(z)})
        %plots a rectangle with its bottom-left vertex at on
        \{z\}, z=0.5; The
%rectangle ends when the next turn-off occurs, and is
0.8 units
%tall.

rectangle('Position', [SC_p.time(ons{controlled(z)})(xx)
controlled(z) - 0.5, SC_p.time(offs{controlled(z)}
(xx)) - SC_p.time(ons{controlled(z)})(xx) , 0.8],
'FaceColor', color(z));

end

datetickzoom('x', 'HH:MM');
xlim([SC_p.time(1), SC_p.time(end)]);
ylim([controlled(1)-1, controlled(end)+1]);
title('HVAC On-Time With Interleaving Control');
xlabel('Time (HH:MM)', 'FontSize', 12);
ylabel('Tent Number', 'FontSize', 12);
box on

ax = gca;
ax.YTickMode = 'manual';
ax.YTick = [controlled(1) controlled(2) controlled(3)
controlled(4)];
linkaxes([a3t, a3b], 'x');

% Examine effect of environmental inputs (wind speed,
environmental temp, and solar radiation) on tent
temperature
if ~isempty(SC_u.Weather.temp.BCIL.time)
    figure()
    leg = cell(1, length(uncontrolled) + 1);
    title('Effect of Environmental Factors on Tent Thermal
         Characteristic');

end
yyaxis left;

a4t = subplot(2,1,1);
for i = 1:length(uncontrolled)
    hold on;
    plot(SC_u.Tent(uncontrolled(i)).return_temp.hobo.time,
         SC_u.Tent(uncontrolled(i)).return_temp.hobo.data);
    leg{i} = [ 'Tent ', num2str(uncontrolled(i)) ];
end
ylabel('Tent Temperature');

yyaxis right;
plot(SC_u.Weather.temp.BCIL.time, SC_u.Weather.temp.BCIL.data);
leg{length(uncontrolled) + 1} = 'Outdoor Temp.';
ylabel('Outside Temperature');
legend(leg);

a4b = subplot(2,1,2);
yyaxis left;
plot(SC_u.Weather.solarrad.BCIL.time, SC_u.Weather.solarrad.BCIL.data);
ylabel('Solar Irradiance');

yyaxis right;
plot(SC_u.Weather.windspeed.BCIL.time, SC_u.Weather.windspeed.BCIL.data);
ylabel('Windspeed');
% Verify calculated on/off times against output temperature data

figure()
hold on;
for i = 1:length(uncontrolled)
    ax5(i) = subplot(length(uncontrolled), 1, i);
    yyaxis left;
    plot(SC_p.time, SC_p.Tent(uncontrolled(i)).output_temp.hobo.data);
    yyaxis right;
    plot(SC_p.time, SC_p.Tent(uncontrolled(i)).scroll_state.controller.data);
end

% Special figure for 02MAR17 dataset: demonstrate failure of independent control scheme to minimize peak load
if strcmp(test_date, 'MAR 02 2017')
    figure()
    subplot(3,1,1);
    plot(SC_p.time, tot_run_u);
    xlabel('Time')
    ylabel('# ECUs Running (out of 4)');
title(['Load Profile with Independent HVAC Operation: ',
    test_date]);
datetickzoom;
xlim([SC_p.time(1), SC_p.time(end)]);
ylim([0, 4.2]);

subplot(3,1,2);
plot(SC_p.time, zeros(1, length(SC_p.time)));
hold on;
for z = 1:length(uncontrolled)
    for xx = 1:length(ons{uncontrolled(z)})
        % plots a rectangle with its bottom-left vertex at
        % on{z}, z - 0.5; The
        % rectangle ends when the next turn-off occurs
        % and is 0.8 units
        % tall.
        rectangle('Position',[SC_p.time(ons{uncontrolled(z)})(xx),
            z - 0.5, SC_p.time(offs{uncontrolled(z)})(xx) - SC_p.time(ons{uncontrolled(z)})(xx),
            0.8], 'FaceColor', color(z));
    end
end
datetickzoom('x', 'HHMM');
xlim([SC_p.time(1), SC_p.time(end)]);
ylim([0, 5]);
title('HVAC On-Time Without Interleaving Control');
xlabel('Time (HHMM)', 'FontSize', 12);
ylabel('Tent Number', 'FontSize', 12);
box on
ax = gca;
ax.YTickMode = 'manual';
ax.YTick = [1 2 3 4];

subplot(3,1,3);
hold on; color = ['r', 'g', 'b', 'k'];
p1b = cell(1,4);
mx = zeros(length(uncontrolled),1);
mm = zeros(length(uncontrolled),1);
for i = 1:length(uncontrolled)
  p1b{i} = plot(SC_p.time, SC_p.Tent(uncontrolled(i)).return_temp.hobo.data, color(i));
  mx(i) = max(SC_p.Tent(uncontrolled(i)).return_temp.hobo.data);
  mm(i) = min(SC_p.Tent(uncontrolled(i)).return_temp.hobo.data);
end
xlabel('Time');
ylabel('Temperature (\circ F)');
title(['Temperature Performance with Independent HVAC Operation: ', test_date]);
datetickzoom;
xlim([SC_p.time(1), SC_p.time(end)]);
ylim([min(mm)-2, max(mx)+2]);
legend('Tent 1', 'Tent 2', 'Tent 3', 'Tent 4');
end
B.4 MATLAB tent thermal modeling

Section 2.1.2 described modeling each group of two adjacent tent sections operating in Heat mode according to a set of fourth-order state equations. The MATLAB tent thermal modeling suite blocks the data generated by compile_data.m (described in Section B.3) into segments comprising a programmable number of heater on/off periods and which are formatted such that the MATLAB function grey_est.m can fit model parameter values for each.

B.4.1 grey_package.m

The function of grey_package.m is to organize the temperature and heater data for each tent complex (internal temperature and heater state of two adjacent tent sections), along with the recorded environmental temperature and solar irradiation, as input and state information according to the formatting specification for MATLAB’s iddata data type. The output of grey_package.m is, for each tent complex, a sequence of iddata variables representing each heater on/off period. The on/off periods are totally disaggregated such that each iddata dataset begins with a heater turn-on or turn-off event and ends on the following heater turn-off or turn-on event, respectively.

```matlab
function [SC_new] = grey_package(SC, start_time, end_time,
                                 bound_method, segment_buffer, inp_res_method, pad)

bound_method = 'mode';
inp_res_method = 'static';

if start_time == -1
    if isempty(SC.Tent(1).return_temp.hobo.time)
        start_time = SC.Tent(1).return_temp.hobo.time(2);
    elseif isempty(SC.Tent(1).return_temp.controller.time)
        start_time = SC.Tent(1).return_temp.controller.time
    else
        start_time = SC.Tent(1).return_temp.controller.time(2);
    end
end
```
(2);

else
    disp('Error: No return temp data in tent object. ');
end

if end-time == -1
    if ~isempty(SC.Tent(1).return_temp.hobo.time)
        end_time = SC.Tent(1).return_temp.hobo.time(end);
    elseif ~isempty(SC.Tent(1).return_temp.controller.time)
        end_time = SC.Tent(1).return_temp.controller.time(end);
    else
        disp('Error: No return temp data in tent object. ');
    end

%% Clip data to argument specifications
hm = pwd;
cd('..//..//2017 Spring')
[SC, ind_start, ind_end] = clip_camp(SC, start_time, end_time);
cd(hm);

%% Create COMPLEX struct, which represents tent couples
complexstruct = struct('iddata_whole', [],...
'iddata_concat', [],...
'iddata_ons', [],...
'iddata_offs', [],...
'model_whole', [],...
'model_concat', [],...
'model_ons', [], ...
'model_offs', []);

Complex = [complex_struct ... %Tents 1 & 2
complex_struct ... %Tents 3 & 4
complex_struct ... %Tents 5 & 6
complex_struct]; %Tents 7 & 8

[SC(:,).Complex] = Complex;

%% Create iddata object for each tent over the specified time interval
len = length(SC.Tent(1).return_temp.hobo.time);

for i = 1:length(SC.Tent)/2
    input = zeros(len, 4);
    output = zeros(len, 2);

    % System Outputs
    if ~isempty(SC.Tent(2*i-1).return_temp.hobo.time)
        s1 = datenum(SC.Tent(2*i-1).return_temp.hobo.time(1));
        s2 = datenum(SC.Tent(2*i-1).return_temp.hobo.time(2));
        Ts = s2(6)-s1(6);
        % Use hobo data
        disp(['Reading hobo data for Tent ', num2str(2*i-1), '. return temp.']);
        output(:,1) = (SC.Tent(2*i-1).return_temp.hobo.data(1):...
elseif ~isempty(SC.Tent(2*i-1).return_temp.controller.time)
    s1 = datevec(SC.Tent(2*i-1).return_temp.controller.time(1));
    s2 = datevec(SC.Tent(2*i-1).return_temp.controller.time(2));
    Ts = s2(6)-s1(6);
    %Use controller data
    disp(['Reading controller data for Tent ', num2str(2*i-1), ', return temp.']);
    output(:,1) = (SC.Tent(2*i-1).return_temp.controller.data-32)/1.8; %Convert to metric

else
    disp(['Error! No return temperature data in Tent ', num2str(2*i-1)]);
    iddata_whole = -1;
    iddata_seg = -1;
    return
end

if ~isempty(SC.Tent(2*i).return_temp.hobo.time)
    s1 = datevec(SC.Tent(2*i).return_temp.hobo.time(1));
    s2 = datevec(SC.Tent(2*i).return_temp.hobo.time(2));
    Ts = s2(6)-s1(6);
    %Use hobo data
    disp(['Reading hobo data for Tent ', num2str(2*i), ', return temp.']);
    output(:,2) = (SC.Tent(2*i).return_temp.hobo.data-32)/1.8; %Convert to metric

270
elseif ~isempty(SC.Tent(2*i).return_temp.controller.time)
    s1 = datevec(SC.Tent(2*i).return_temp.controller.time(1));
    s2 = datevec(SC.Tent(2*i).return_temp.controller.time(2));
    Ts = s2(6)-s1(6);
    %Use controller data
    disp(['Reading controller data for Tent ', num2str(2*i), ' return temp.']);
    output(:,2) = (SC.Tent(2*i).return_temp.controller.data-32)/1.8; %Convert to metric
else
    disp(['Error! No return temperature data in Tent ', num2str(2*i)]);
    iddata_whole = -1;
    iddata_seg = -1;
    return
end

%System Inputs
input(:,1) = (SC.Weather.temp.BCIL.data-32)/1.8;
input(:,2) = SC.Weather.solarrad.BCIL.data;

if strcmp(inp_res_method, 'static')
    if ~isempty(SC.Tent(2*i-1).mode.controller.time)
        %Use controller data
        disp(['Reading controller data for Tent ', num2str(2*i-1), ' mode.']);
        input(:,3) = (SC.Tent(2*i-1).mode.controller.data-1)*9e3;
    end
end
elseif ~isempty(SC.Tent(2*i-1).mode.hobo.time)
    %Use hobo data
    disp(['Reading hobo data for Tent ', num2str(2*i -1), ' mode.'']);
    input(:,3) = (SC.Tent(2*i-1).mode.hobo.data -1)*9
e3;
else
    disp(['Error! No mode data in Tent ', num2str(2*i -1)]);
end

if ~isempty(SC.Tent(2*i).mode.controller.time)
    %Use controller data
    disp(['Reading controller data for Tent ',
         num2str(2*i), ' mode.']);
    input(:,4) = (SC.Tent(2*i).mode.controller.data
                    -1)*.9e4;
elseif ~isempty(SC.Tent(2*i).mode.hobo.time)
    %Use hobo data
    disp(['Reading hobo data for Tent ', num2str(2*i)
          , ' mode.']);
    input(:,4) = (SC.Tent(2*i).mode.hobo.data -1)*.9e4
    ;
else
    disp(['Error! No mode data in Tent ', num2str(2*i ))]);
end

elseif strcmp(inp_res_method, 'temp')
    if ~isempty(SC.Tent(2*i-1).output_temp.hobo.time)
%Use hobo data

disp(['Reading hobo data for Tent ', num2str(2*i-1), ' output temp.']);

if ~isempty(SC.Tent(2*i-1).return_temp.hobo.time)
    disp(['Reading hobo data for Tent ', num2str(2*i-1), ' return temp.']);
    input(:,3) = (SC.Tent(2*i-1).output_temp.hobo.data-32)/1.8; %Convert to metric, apply estimated 180 Tdiff->Pin constant

elseif ~isempty(SC.Tent(2*i-1).return_temp.controller.time)
    disp(['Reading controller data for Tent ', num2str(2*i-1), ' return temp.']);
    input(:,3) = (SC.Tent(2*i-1).output_temp.hobo.data-32)/1.8;

else
    disp(['Error! No return temperature data for Tent ', num2str(2*i-1)]);
end

elseif ~isempty(SC.Tent(2*i-1).output_temp.controller.time)
    if ~isempty(SC.Tent(2*i-1).return_temp.hobo.time)
        disp(['Reading hobo data for Tent ', num2str(2*i-1), ' return temp.']);
        input(:,3) = (SC.Tent(2*i-1).output_temp.controller.data-32)/1.8; %Convert to metric, apply estimated 180 Tdiff->Pin constant

    elseif ~isempty(SC.Tent(2*i-1).return_temp.controller.time)

disp(['Reading controller data for Tent ',
num2str(2*i-1), ' return temp.']);
input(:,3) = (SC.Tent(2*i-1).output_temp.
controller.data-32)/1.8;
else
disp(['Error! No return temperature data for
Tent ', num2str(2*i-1)]);
end
derm

disp(['Error!
No output temperature data for Tent
', num2str(2*i-1)]);
end
if ~isempty(SC.Tent(2*i).output_temp.hobo.time)
  disp(['Reading hobo data for Tent ',
num2str(2*i), ' output temp.']);
  if ~isempty(SC.Tent(2*i-1).return_temp.hobo.time)
    disp(['Reading hobo data for Tent ',
num2str(2*i), ' return temp.']);
    input(:,3) = (SC.Tent(2*i).output_temp.hobo.
data-32)/1.8; %Convert to metric, apply
    estimated 180 Tdiff→Pin constant
  elseif ~isempty(SC.Tent(2*i).return_temp.
controller.time)
    disp(['Reading controller data for Tent ',
num2str(2*i), ' return temp.']);
    input(:,3) = (SC.Tent(2*i).output_temp.hobo.
data-32)/1.8;
  else
disp(['Error! No return temperature data for
Tent ', num2str(2*i)];
end

Tent ', num2str(2*i)]

elseif ~isempty(SC.Tent(2*i).output_temp.controller.time)

    if ~isempty(SC.Tent(2*i).return_temp.hobo.time)
        disp(['Reading hobo data for Tent ', num2str(2*i), ' return temp.']);
        input(:,4) = (SC.Tent(2*i).output_temp.controller.data-32)/1.8; %Convert to metric, apply estimated 180 Tdiff—>Pin constant
    elseif ~isempty(SC.Tent(2*i).return_temp.controller.time)
        disp(['Reading controller data for Tent ', num2str(2*i), ' return temp.']);
        input(:,4) = (SC.Tent(2*i).output_temp.controller.data-32)/1.8;
    else
        disp(['Error! No return temperature data for Tent ', num2str(2*i)]);
    end
else
    disp(['Error! No output temperature data for Tent ', num2str(2*i)]);
end
end

SC.Complex(i).iddata_whole = iddata(output, input, Ts);
end

275
Segment iddata objects for individual fits

% Find indices of turn-on and turn-off events for each
% COMPLEX, using the
% SOUTHWARD tents! (odd-numbered).

for i = 1:4
    % The iterative way
    if ~isempty(SC.Tent(2*i-1).mode.controller.time)
        turn_on = [0; diff(SC.Tent(2*i-1).mode.controller.
                   data) == 1];
        turn_off = [0; diff(SC.Tent(2*i-1).mode.controller.
                    data) == -1];
    elseif ~isempty(SC.Tent(2*i-1).mode.hobo.time)
        turn_on = [0; diff(SC.Tent(2*i-1).mode.hobo.data) == 1];
        turn_off = [0; diff(SC.Tent(2*i-1).mode.hobo.data) == -1];
    else
        disp('Caught ya!');
    end

    if find(turn_on, 1, 'last') > find(turn_off, 1, 'last')
        turn_off(end) = 1;
    end

    if find(turn_off, 1, 'first') < find(turn_on, 1, 'first')
        turn_on(1) = 1;
    end

    % Make vectors that contain the indices of each turn-on
and turn-off
ons = find(turn_on);
offs = find(turn_off);

% Check that some turn-on and turn-off events were
detected.
% Ensure that first event is a turn-on and last event is
a turn-off
if sum(turn_on) > 0 && sum(turn_off) > 0
    if ons(1) > offs(1)
        ons = [1, ons];
    end
    if offs(end) < ons(end)
        offs = [offs, length(turn_off)];
    end
else
    disp('Error! The event-finding procedure failed. ');
disp('Error type: no turn-on events detected, and/or
    no turn-off events detected. ');
    error_code = 1;
    return;
end

% Check that turn-ons and turn-offs strictly alternate
if length(ons) == length(offs)
    if sum(offs-ons > 0) == length(ons)
        disp('Success! Turn-on and turn-off events
            strictly alternate. ');
    else
        disp('Error! Turn-on and turn-off events do not
strictly alternate.');
    disp('Error type: multiple consecutive turn-on and turn-off events.);
    error_code = 2;
    return;
  end
else
    disp('Error! Turn-on and turn-off events do not strictly alternate.);
    disp('Error type: unequal number of turn-on and turn-off events.);
    error_code = 3;
    return;
end

% Separate response data for all turn-on and turn-off events
on_events = cell(1,length(ons));
off_events = on_events;

% make an iddata object for each turn-on event
for q = 1:length(ons)
    output = SC.Complex(i).
              iddata_whole.OutputData(ons(q)+segment_buffer:offs(q)-segment_buffer,:);

    input = SC.Complex(i).iddata_whole.
            InputData(ons(q)+segment_buffer:offs(q)-segment_buffer,:);

    % [input, output] = grey_pad(input, output);
on_events{q}.ID = iddata(output, input, Ts);
on_events{q}.RealInput = SC.Complex(i).iddata_whole.
    InputData(ons(q)+segment_buffer:offs(q)-
    segment_buffer,3);
on_events{q}.start = ons(q) + segment_buffer;
on_events{q}.end = offs(q) - segment_buffer;
end

%make an iddata object for each turn-off event
for q = 1:length(ons)-1
    output = SC.Complex(i).iddata_whole.
        OutputData(offs(q)+segment_buffer+1:ons(q+1)-
        segment_buffer-1,:);
    input = SC.Complex(i).iddata_whole.InputData(offs(q)+
        segment_buffer+1:ons(q+1)-segment_buffer-1,:);
    %
    [input, output] = grey_pad(input, output);
    off_events{q}.ID = iddata(output, input, SC.
        Complex(i).iddata_whole.Ts);
    off_events{q}.RealInput = SC.Complex(i).iddata_whole.
        InputData(offs(q)+segment_buffer+1:ons(q+1)-
        segment_buffer-1,3);
    off_events{q}.start = offs(q) + segment_buffer
        + 1;
    off_events{q}.end = ons(q+1) - segment_buffer
        - 1;
end

%create an iddata object for the final turn-off event
if offs(end) == length(SC.Complex(i).iddata Whole.
  OutputData)
    off_events = off_events(1:end-1);
% output
    = zeros(4,1);
% input
    = zeros(4,3);
% % [input, output] = grey_pad(input,
    output);
% off_events{end}.ID
    = iddata(output, input,
    SC.Tent(i).iddata Whole.Ts);
% off_events{end}.RealInput
    = zeros(4,3);
% off_events{end}.start = offs(end-1) +
    segment_buffer;
% off_events{end}.end = ons(end) -
    segment_buffer;
else
  output
    = SC.Complex(i).
    iddata Whole.OutputData(offs(end)+1:end,:);
  input
    = SC.Complex(i).
    iddata Whole.InputData(offs(end)+1:end,:);
  % [input, output] = grey_pad(input, output)
  ;
    off_events{end}.ID
    = iddata(output, input, SC.
    Complex(i).iddata Whole.Ts);
    off_events{end}.RealInput
    = SC.Complex(i).
    iddata Whole.InputData(offs(end)+1:end,:);
    off_events{end}.start
    = offs(end) +
    segment_buffer + 1;
    off_events{end}.end = length(SC.Complex(i).
    iddata Whole.OutputData);
    if off_events{end}.end - off_events{end}.start + 1 ~=

length (off_events {end} . ID . OutputData )

pass

end

end

SC . Complex ( i ) . iddata_ons = on_events ;
SC . Complex ( i ) . iddata_offs = off_events ;

end

SC_new = SC;

B.4.2 grey_concat_2 . m

grey_concat_2 . m combines the individual heater on/off iddata datasets generated by grey_package . m for each tent complex into longer datasets so that the model can be fit over a controllable number of heater on/off periods. Its concatenation mode can be set to " individual," which has it combine adjacent iddata datasets such that the model is fit to a heater on and a heater off period for each tent complex, " concat" of degree $n$, which has it combine $n$ adjacent heater on/off periods before fitting the model, or " whole" which has it recombine all of the input and state data for each tent so that a model is fit to the entire dataset.

function SC = grey_concat_2 (SC, G, pad)

% You have been passed N "on" iddata objects and N or N-1 "off" iddata objects.
% Your task is to concatenate them into groups of G on-off sets. If N/G is
% noninteger, then the final set will consist of $N / G$ on-off sets. If there
% are not N but rather N-1 "off" iddata objects,
% Do this for each Tent...
for i = 1:4
    N = length(SC.Complex(i).iddata_ons);
    fraglen = mod(N, G); % # of on-off sets in final concatenated set. 0 if N divides evenly into G sets
    fragged = (fraglen ~= 0); % 1 if N/G is noninteger, 0 if it is.
    numsets = floor(N/G) + fragged; % # of concatenated sets to be generated, including the set fragment

    % Generate the full concatenated sets
    for q = 1:numsets-1
        setlength = 0;

        start = SC.Complex(i).iddata_ons{(q-1)*G+1}.start; % Record the start and finish indices of the concatenated dataset,
        finish = SC.Complex(i).iddata_offs{(q)*G}.end;
        % relative to the test timeseries.

        for s = 1:G % count length of G on-off sets
            setlength = setlength + length(SC.Complex(i).
                                           iddata_ons{(q-1)*G+s}.ID.OutputData);
            setlength = setlength + length(SC.Complex(i).
                                           iddata_offs{(q-1)*G+s}.ID.OutputData);
        end

        input = zeros(setlength, 4); output = zeros(setlength, 2); % Make empty input/output vectors
ind_st = 1;

Set starting index for populating input/output vectors

for s = 1:G %put together G on-off sets
    fin = ind_st + length(SC.Complex(i).iddata_ons{(q-1)*G+s}.ID.OutputData) - 1;
    input(ind_st:fin,:) = SC.Complex(i).iddata_ons{(q-1)*G+s}.ID.InputData;
    output(ind_st:fin,:) = SC.Complex(i).
        iddata_ons{(q-1)*G+s}.ID.OutputData;
    ind_st = fin + 1;

    fin = ind_st + length(SC.Complex(i).iddata_offs{(q-1)*G+s}.ID.OutputData) - 1;
    input(ind_st:fin,:) = SC.Complex(i).iddata_offs{(q-1)*G+s}.ID.InputData;
    output(ind_st:fin,:) = SC.Complex(i).
        iddata_offs{(q-1)*G+s}.ID.OutputData;
    ind_st = fin + 1;
end

SC.Complex(i).idata_concat{q}.ID = iddata(output,
    input, SC.Complex(i).idata_whole.Ts);
SC.Complex(i).idata_concat{q}.start = start;
SC.Complex(i).idata_concat{q}.end = finish;
end

%% Generate the final concatenated set, with provisions
for 1) noninteger N/G, 2) inequal # of on/off events

define variables:

q = numsets;
setlength = 0;

start = SC.Complex(i).iddata_ons{(q-1)*G+1}.start;   %
Record the start and finish indices of the
concatenated dataset,

if length(SC.Complex(i).iddata_offs) == length(SC.Complex
(i).iddata_ons)
  if fragged == 0
    finish = SC.Complex(i).iddata_offs{(q*G)}.end;   %
    There is a final off dataset
  else
    finish = SC.Complex(i).iddata_offs{(q-1)*G+
        frag_len}.end;
  end
else
  if fragged == 0
    finish = SC.Complex(i).iddata_offs{(q*G)}.end;   %
    Final "off" event happens at the end of the
    dataset; no additional data!
  else
    finish = SC.Complex(i).iddata_offs{(q-1)*G+
        frag_len}.end;
  end
end

if fragged == 0
  for s = 1:G-1 %count length of frag_len on-off sets
    setlength = setlength + length(SC.Complex(i).

iddata_ons{((q-1)*G+s).ID.OutputData};
setlength = setlength + length(SC.Complex(i).
iddata_ons{((q-1)*G+s).ID.OutputData};
end
s = G;
else
for s = 1:frag_len-1 %count length of frag_len on-off
sets
setlength = setlength + length(SC.Complex(i).
iddata_ons{((q-1)*G+s).ID.OutputData};
setlength = setlength + length(SC.Complex(i).
iddata_offs{((q-1)*G+s).ID.OutputData};
end
s = frag_len;
end

%Add the length of the final "on" dataset
setlength = setlength + length(SC.Complex(i).iddata_ons{((q-1)*G+s).ID.OutputData};

%The final "off" dataset may not exist. If it does, add its length
if length(SC.Complex(i).iddata_offs) == length(SC.Complex(i).iddata_ons)
setlength = setlength + length(SC.Complex(i).
iddata_offs{((q-1)*G+s).ID.OutputData};
end

input = zeros(setlength, 4); output = zeros(setlength, 2)
; %Make empty input/output vectors
ind_st = 1;                        %Set
starting index for populating input/output vectors

if fragged == 0
    for s = 1:G-1                 %put together G on-off sets
        fin = ind_st + length(SC.Complex
                          (i).iddata_ons{(q-1)*G+s}.ID.OutputData) - 1;
        input(ind_st:fin,:) = SC.Complex(i).iddata_ons{
                           (q-1)*G+s}.ID.InputData;
        output(ind_st:fin,:) = SC.Complex(i).
                           iddata_ons{(q-1)*G+s}.ID.OutputData;
        ind_st = fin + 1;

        fin = ind_st + length(SC.Complex
                          (i).iddata_offs{(q-1)*G+s}.ID.OutputData) - 1;
        input(ind_st:fin,:) = SC.Complex(i).iddata_offs{
                           (q-1)*G+s}.ID.InputData;
        output(ind_st:fin,:) = SC.Complex(i).
                           iddata_offs{(q-1)*G+s}.ID.OutputData;
        ind_st = fin + 1;
    end
    s = G;
else
    for s = 1:frag_len-1          %put together frag_len on-off sets
        fin = ind_st + length(SC.Complex
                          (i).iddata_ons{(q-1)*G+s}.ID.OutputData) - 1;
        input(ind_st:fin,:) = SC.Complex(i).iddata_ons{
                           (q-1)*G+s}.ID.InputData;
    end
end
output(ind_st:fin,:) = SC.Complex(i).
iddata_ons{(q-1)*G+s}.ID.OutputData;

ind_st = fin + 1;

fin = ind_st + length(SC.Complex(i).iddata_offs{(q-1)*G+s}.ID.OutputData) - 1;
input(ind_st:fin,:) = SC.Complex(i).iddata_offs{(q-1)*G+s}.ID.InputData;
output(ind_st:fin,:) = SC.Complex(i).
iddata_offs{(q-1)*G+s}.ID.OutputData;
ind_st = fin + 1;
end

s = frag_len;
end

%Add the final on dataset

fin = ind_st + length(SC.Complex(i).
iddata_ons{(q-1)*G+s}.ID.OutputData) - 1;
input(ind_st:fin,:) = SC.Complex(i).iddata_ons{(q-1)*G+s}.ID.InputData;
output(ind_st:fin,:) = SC.Complex(i).iddata_ons{(q-1)*G+s}.ID.OutputData;
ind_st = fin + 1;

%Add the final off dataset, if it exists

if length(SC.Complex(i).iddata_offs) == length(SC.Complex(i).iddata_ons)
fin = ind_st + length(SC.Complex(i).
iddata_offs{(q-1)*G+s}.ID.OutputData) - 1;
input(ind_st:fin,:) = SC.Complex(i).iddata_offs{(q-1)*G+s}.ID.InputData;
else
input(ind_st:fin,:) = SC.Complex(i).iddata_offs{(q-1)*G+s}.ID.InputData;
output(ind_st:fin,:) = SC.Complex(i).iddata_offs{(q-1)*G+s}.ID.OutputData;
ind_st = fin + 1;
end
-1)*G+s}.ID.InputData;

output(ind_st:fin,:) = SC.Complex(i).iddata_offs{((q-1)*G+s}).ID.OutputData;

end

SC.Complex(i).iddata_concat{q}.ID = iddata(output, input, SC.Complex(i).iddata_whole.Ts);
SC.Complex(i).iddata_concat{q}.start = start;
SC.Complex(i).iddata_concat{q}.end = finish;

if SC.Complex(i).iddata_concat{q}.end + 1 - SC.Complex(i)
 .iddata_concat{q}.start ~= length(SC.Complex(i).
   iddata_concat{q}.ID.OutputData)
   disp('error!')
end
end

B.4.3 grey_fit.m

grey_fit.m takes an iddata dataset generated for a tent complex by concat_data_2.m
and uses it to fit parameter values to the model specified by grey_tent_1D.m.

function SC_n = grey_fit(SC, min_len, inp_del, fit_mode,
   flow_est, pad, est)

  % set options for grey-box model fitting
  optest = greyestOptions('InitialState','backcast');

  % Create initial system guess

  % Estimate initial parameter vector

288
tau = 5*60;  % Time constant estimate from experimental data.
Rte = 0.0018296;%0.0015;%0.0036;   % Thermal resistance estimate from tent manufacturer.
Rtt = 0.01;%0.0057;%0.002;%
C = 380000;%1003*550*1.225; % Estimated value of thermal capacitance, based on R and tau.
kS = 10;%15;%0.07;
kN = 10;%15;
par = [Rte, C, kS, kN, flow_est, Rtt, Rte, C, flow_est]; %[1/
tau; 0*0.07/C; 1/C]; % initial parameter vector —ALT— [R, C, k]
p = cell(1,4);%

20 %22MAY 1600—2200
21 p{1} = [.00176 404426 14.18 9.82 .018 .00288 .00227 387346 .019];
22 p{2} = [.00223 346422 12.50 11.7 .023 .00396 .00181 367981 .016];
23 p{3} = [.00211 436906 19.00 16.1 .011 .00237 .00190 332428 .012];
24 p{4} = [.00186 422131 22.10 15.8 .011 .00295 .00192 444125 .013];

aux = {};  % additional arguments passed to function (not used)
T = 0;      % set to 0 because a continuous model
%Create estimated system with physical constraints
m = idgrey('grey_tent_1D',par,'c',aux,T,'Name','1-D Tent Model');

m.Structure.Parameters.Minimum(1) = Rte/2;
m.Structure.Parameters.Minimum(2) = C/2;
m.Structure.Parameters.Minimum(3) = 0;
m.Structure.Parameters.Minimum(4) = 0;
m.Structure.Parameters.Minimum(5) = flow_est/2;
m.Structure.Parameters.Minimum(6) = Rtt/20;
m.Structure.Parameters.Minimum(7) = Rte/2;
m.Structure.Parameters.Minimum(8) = C/2;
m.Structure.Parameters.Minimum(9) = flow_est/2;

m.Structure.Parameters.Maximum(1) = Rte*2;
m.Structure.Parameters.Maximum(2) = C*2;
m.Structure.Parameters.Maximum(3) = kS*4;
m.Structure.Parameters.Maximum(4) = kN*4;
m.Structure.Parameters.Maximum(5) = flow_est*2;
m.Structure.Parameters.Maximum(6) = Rtt*50;
m.Structure.Parameters.Maximum(7) = Rte*2;
m.Structure.Parameters.Maximum(8) = C*2;
m.Structure.Parameters.Maximum(9) = flow_est*2;

m.InputDelay = [0;0;inp_del;inp_del];

%% conduct model-fitting
grey_mod = struct('Model', 0, 'params', 0, 'uncertainty', 0,
for i = 1:4
    SC.Complex(i).model_whole = grey_mod;
    if length(SC.Complex(i).iddata_whole.OutputData) >= min_len
        if est == 1
            SC.Complex(i).model_whole.Model =
            greyest(SC.Complex(i).iddata_whole, m, optest);
            [SC.Complex(i).model_whole.params, SC.Complex(i).
            model_whole.uncertainty] = getpvec(SC.Complex(i).
            model_whole.Model);
            SC.Complex(i).model_whole.pts =
            length(SC.Complex(i).iddata_whole.OutputData);
        else
            par = p{i};
            m = idgrey('grey_tent_1D', par, 'c', aux, T, 'Name',
            '1-D Tent Model');
            SC.Complex(i).model_whole.Model = m;
            [SC.Complex(i).model_whole.params, SC.Complex(i).
            model_whole.uncertainty] = getpvec(m);
            SC.Complex(i).model_whole.pts =
            length(SC.Complex(i).iddata_whole.OutputData);
        end
    end
end
if strcmp(fit_mode, 'individual')
    on_models = cell(1, length(SC.Complex(i).iddata_ons));
    for q = 1:length(SC.Complex(i).iddata_ons)
        on_models{q} = grey_mod;
        if length(SC.Complex(i).iddata_ons{q}.ID. OutputData) >= min_len
            disp(['Complex ', num2str(i), ', dataset ', num2str(q), ', turn-on']);
            on_models{q}.Model = greyest(SC.Complex(i).iddata_ons{q}.ID, m, optest);
        end
    end
    off_models = cell(1, length(SC.Complex(i).iddata_offs));
    for q = 1:length(SC.Complex(i).iddata_offs)
        off_models{q} = grey_mod;
        if length(SC.Complex(i).iddata_offs{q}.ID. OutputData) >= min_len
            disp(['Complex ', num2str(i), ', dataset ', num2str(q), ', turn-off']);
            off_models{q}.Model = length(SC.Complex(i).iddata_offs{q}.ID. OutputData);
        end
    end
= greyest(SC.Complex(i).iddata_offs{q}.ID, m, optest);

[off_models{q}.params, off_models{q}.uncertainty] = getpvec(off_models{q}.Model);

off_models{q}.pts

                = length(SC.Complex(i).iddata_offs{q}.ID.OutputData);

SC.Complex(i).off_models{q}

                = off_models{q};

end

end

if strcmp(fit_mode, 'concat')

    concat_models = cell(1,length(SC.Complex(i).iddata_concat));

    for q = 1:length(SC.Complex(i).iddata_concat)

        if length(SC.Complex(i).iddata_concat{q}.ID.OutputData) >= min_len

            output = SC.Complex(i).

                       iddata_concat{q}.ID.OutputData;

            input

                       = SC.Complex(i).

                       iddata_concat{q}.ID.InputData;

            if ~isempty(SC.Tent(2*i-1).output_temp.hobo.data)

                pwr_S_est

                = (SC.Tent(2*i-1).

                           output_temp.hobo.data(SC.Complex(i).
iddata_concat{q}.start) - SC.Tent(2*i -1).return_temp.hobo.data(SC.Complex(i).iddata_concat{q}.start)) *200/1.8;

else

    pwr_S_est = (SC.Tent(2*i-1).
    output_temp.controller.data(SC.Complex(i).iddata_concat{q}.start) - SC.Tent(2*i-1).return_temp.hobo.data(SC.Complex(i).iddata_concat{q}.start)) *200/1.8;

end

if ~isempty(SC.Tent(2*i).output_temp.hobo.data)

    pwr_N_est = (SC.Tent(2*i).
    output_temp.hobo.data(SC.Complex(i).iddata_concat{q}.start) - SC.Tent(2*i).return_temp.hobo.data(SC.Complex(i).iddata_concat{q}.start)) *200/1.8;

else

    pwr_N_est = (SC.Tent(2*i).
    output_temp.controller.data(SC.Complex(i).iddata_concat{q}.start) - SC.Tent(2*i).return_temp.hobo.data(SC.Complex(i).iddata_concat{q}.start)) *200/1.8;

end

id_temp = iddata(output, input,
SC.Complex(i).iddata_whole.Ts);

optest = greyestOptions('InitialState', [}
output(1,1); pwr_S_est; output(1,2); pwr_N_est));
disp(['Complex ', num2str(i), ', dataset ', num2str(q), ', concatenated']);
concat_models{q}.Model = greyest (id_temp, m, optest);
[concat_models{q}.params, concat_models{q}.uncertainty] = getpvec(concat_models{q}.Model);
concat_models{q}.pts = length(SC.Complex(i).iddata_concat{q}.ID.OutputData);
SC.Complex(i).concat_models{q} = concat_models{q};
end
end
end

end

SC_n = SC;

B.4.4  grey_eval.m

grey_eval.m compares the response of the models generated by grey_fit.m to the iddata sets (created by grey_concat_2.m) that were used to generate the fits. It can be programmed to only show model fits that were better or worse than a pro-
grammable threshold according to MATLAB's "goodness of fit" criterion. Therefore, grey_eval_2.m can be used to view all model fits, only the best model fits, or alternatively only the worst model fits.

```matlab
function [SC_n, success, avg_err, max_err, min_err, abs_err] = grey_eval(SC, eval_mode, min_good, g, model_extend, preferential)

plotnum = 1;
left_color = [0 0 0];
right_color = [.75 .75 1];

if strcmpi(eval_mode, 'whole');
    success = zeros(1, 8);

    for i = 1:4
        [y, fit, x0] = compare(SC.Complex(i) .iddata_whole, SC .Complex(i) .model_whole.Model);

        outp_S = y .OutputData(:,1);
        outp_N = y .OutputData(:,2);

        fit_S = fit (1);
        fit_N = fit (2);

        avg_err{1}(1, 2*i-1) = mean(outp_S) - mean(SC.Complex(i) .iddata_whole .OutputData(:,1));
        max_err{1}(1, 2*i-1) = max(outp_S) - max(SC.Complex(i) .iddata_whole .OutputData(:,1));
        min_err{1}(1, 2*i-1) = min(outp_S) - min(SC.Complex(i) .iddata_whole .OutputData(:,1));
```
\[
\text{abs\_err} \{1\}(1, 2i-1) = \max(\text{abs}(\text{outp\_S} - \text{SC\_Complex}(i) \text{. iddata\_whole\_OutputData(:,1)))};
\]
\[
\text{avg\_err} \{1\}(1, 2i) = \text{mean}(\text{outp\_N}) - \text{mean}(\text{SC\_Complex}(i) \text{. iddata\_whole\_OutputData(:,2)}); \]
\[
\text{max\_err} \{1\}(1, 2i) = \max(\text{outp\_N}) - \max(\text{SC\_Complex}(i) \text{. iddata\_whole\_OutputData(:,2)}); \]
\[
\text{min\_err} \{1\}(1, 2i) = \min(\text{outp\_N}) - \min(\text{SC\_Complex}(i) \text{. iddata\_whole\_OutputData(:,2)}); \]
\[
\text{abs\_err} \{1\}(1, 2i) = \max(\text{abs}(\text{outp\_N} - \text{SC\_Complex}(i) \text{. iddata\_whole\_OutputData(:,2))}); \]

\[
\text{if } \text{g\_fit\_S} > \text{g\_min\_good\%} && \text{sum}([\text{SC\_Complex}(i). \text{model\_whole\_params}(1) < 0, \text{SC\_Complex}(i). \text{model\_whole\_params}(1) < 0]) = 0
\]
\[
\text{success}(1, 2i-1) = \text{success}(1, 2i-1) + 1/\text{length}(\text{SC\_Complex}(i). \text{model\_whole}); \]
\[
\text{fig} = \text{figure}(\text{plotnum});
\]
\[
\% \text{set}(\text{fig},'\text{defaultAxesColorOrder}',[\text{left\_color}; \text{right\_color}]); \]
\[
\text{if } \sim\text{isempty}(\text{SC\_Tent}(2i-1). \text{return\_temp\_controller} \text{. time})
\]
\[
\%\text{Use controller data}
\]
\[
\text{time} = \text{SC\_Tent}(2i-1). \text{return\_temp\_controller} \text{. time};
\]
\[
\text{elseif } \sim\text{isempty}(\text{SC\_Tent}(2i-1). \text{return\_temp\_hobo} \text{. time})
\]

297
%Use hobo data

time = SC.Tent(2*i-1).return_temp.hobo.time;

else

disp(['Error! No return temp data in Tent ', num2str(i)]);

end

subplot(3,3,[1,2,3]);

yyaxis left;

plot(time, SC.Complex(i).iddata_whole.OutputData(:,1), 'k');
hold on;
plot(time, outp_S, 'r-');
legend('Measured Response', ['Model Response', ''], num2str(fit_S), '%');
plotnum = plotnum + 1;
title(['Tent ', num2str(2*i-1), ' R_t_e = ', num2str(SC.Complex(i).model_whole.params(1)), ', C = ', num2str(SC.Complex(i).model_whole.params(2)), ', k = ', num2str(SC.Complex(i).model_whole.params(3)), ', \alpha = ', num2str(SC.Complex(i).model_whole.params(5)), ', R_T_T = ', num2str(SC.Complex(i).model_whole.params(6))]);
ylabel('Temperature (^oC)');

yyaxis right;

plot(time, SC.Complex(i).iddata_whole.InputData(:,3));
datetickzoom
subplot(3,3,[4,5,6]);
yyaxis left;
plot(time, SC.Complex(i).iddata_whole.InputData(:,1));
ylabel('Outdoor Temp');
yyaxis right;
plot(time, SC.Complex(i).iddata_whole.InputData(:,2));
ylabel('Solar Irradiance');
datetickzoom

subplot(3,3,[7,8,9]);
yyaxis left;
plot(time, SC.Weather.windspeed.BCIL.data);
ylabel('Windspeed');
xlabel('Time');
datetickzoom

end

if g*fit_N > g*min_good% && sum([SC.Complex(i).model_whole.params(1) < 0, SC.Complex(i).model_whole.params(1) < 0]) == 0
success(1, 2*i) = success(1, 2*i) + 1/length(SC.Complex(i).model_whole);
fig = figure(plotnum);

set(fig, 'defaultAxesColorOrder', [left_color; right_color]);
if ~isempty(SC.Tent(2*i).return_temp.controller.time)
    %Use controller data
    time = SC.Tent(2*i).return_temp.controller.time;
else if ~isempty(SC.Tent(2*i).return_temp.hobo.time)
    %Use hobo data
    time = SC.Tent(2*i).return_temp.hobo.time;
else
    disp(['Error! No return temp data in Tent ', num2str(i)]);
end

subplot(3,3,[1,2,3]);
yyaxis left;
plot(time, SC.Complex(i).iddata.whole.OutputData(:,2), 'k');
hold on;
plot(time, outp_N, 'r-');
title([['Tent ', num2str(2*i), ' R_t_e = ', num2str(SC.Complex(i).model_whole.params(7)), ' C = ', num2str(SC.Complex(i).model_whole.params(8)), ' k = ', num2str(SC.Complex(i).model_whole.params(4)), '\alpha = ', num2str(SC.Complex(i).model_whole.params(9)), ' R_T_T = ', num2str(SC.Complex(i).model_whole.params(6))]])

legend('Measured Response', ['Model Response', num2str(fit), '%'])
plotnum = plotnum + 1;

ylabel('Temperature (°C)');

plot(time, SC.Complex(i).iddata_whole.InputData (:,4));

datetickzoom

subplot(3,3,[4,5,6]);

plot(time, SC.Complex(i).iddata_whole.InputData (:,1));

ylabel('Outdoor Temp');

yyaxis right;

plot(time, SC.Complex(i).iddata_whole.InputData (:,2));

ylabel('Solar Irradiance');

yyaxis left;

subplot(3,3,[7,8,9]);

plot(time, SC.Weather.windspeed.BCIL.data);

ylabel('Windspeed');

yyaxis right;

plot(time, SC.Weather.humidity.BCIL.data);

ylabel('Humidity');

xlabel('Time');

datetickzoom

end
end

elseif strcmpi(eval_mode, 'individual')
on_success = zeros(1, 8);
off_success = zeros(1, 8);

for i = 1:4
    num_onplots = 0;
    num_offplots = 0;

    avg_err = {zeros(1, 8, length(SC.Complex(i).on_models)-model_extend), zeros(1, 8, length(SC.Complex(i).on_models)-model_extend)};
    max_err = avg_err;
    min_err = avg_err;

    last_good_on_model = [];
    last_good_off_model = [];

    for n = 1:length(SC.Complex(i).on_models)-model_extend
        if ~isempty(SC.Complex(i).on_models{n})
            [y, SC.Complex(i).on_models{n}.fit, x0] = compare(SC.Complex(i).iddata_ons{n+model_extend}.ID, SC.Complex(i).on_models{n}.Model);

            outp_S = y.OutputData(:,1);
            outp_N = y.OutputData(:,2);
            fit_S = fit(1);
            fit_N = fit(2);
if ~isempty (last_good_on_model) && preferential == 1

[y_o, fit, x0_o] = compare(SC.Complex(i).iddata_ons{n+model_extend}.ID, last_good_on_model);

if fit > SC.Complex(i).on_models{n}.fit
    SC.Complex(i).on_models{n}.Model = last_good_on_model;
    y = y_o;
    x0 = x0_o;
else
    last_good_on_model = SC.Complex(i).on_models{n}.Model;
end

elseif g*SC.Complex(i).on_models{n}.fit > g*min_good
    last_good_on_model = SC.Complex(i).on_models{n}.Model;
end

avg_err{1}(1, i, n) = mean(y.OutputData) - mean(SC.Complex(i).iddata_ons{n+model_extend}.ID.OutputData);
max_err{1}(1, i, n) = max(y.OutputData) - max(SC.Complex(i).iddata_ons{n+model_extend}.ID.OutputData);
min_err{1}(1, i, n) = min(y.OutputData) - min(SC.Complex(i).iddata_ons{n+model_extend}.ID.OutputData);
if g*SC.Complex(i).on_models{n}.fit > g*
in_good% && sum([SC.Complex(i).
model_whole.params(1) < 0, SC.Complex(i).
model_whole.params(3) < 0]) == 0
on_success(1, i) = on_success(1, i) + 1/
length(SC.Complex(i).on_models);
fig = figure(plotnum);
set(fig,'defaultAxesColorOrder',[left_color; right_color]);
time = SC.Weather.temp.BCIL.time(SC.
Complex(i).iddata_ons{n+model_extend}).
start:SC.Complex(i).iddata_ons{n+
model_extend}.end);

subplot(2,3,[1,2,3]);
yyaxis left;
plot(time, SC.Complex(i).iddata_ons{n+model_extend}.ID.OutputData, 'k');
hold on;
plot(time, y.ID.OutputData, 'r--');
datetickzoom
yyaxis right;
plot(time, SC.Complex(i).iddata_ons{n+model_extend}.RealInput);

ID.InputData(:,3));

title(['Complex ', num2str(i), ' R ='])
num2str(SC.Complex(i).on_models{n}.params(1)), ' C = ', num2str(SC.Complex(i).on_models{n}.params(2)), ' k = ', num2str(SC.Complex(i).on_models{n}.params(3)), ' \alpha = ', num2str(SC.Complex(i).on_models{n}.params(4)))

legend('Measured Response (on)', ['Model Response', ' C = ', num2str(SC.Complex(i).on_models{n}.fit), '%', 'Input (ECU Supply temp.)']);
datetickzoom

[y, t] = step(SC.Complex(i).on_models{n+model_extend}.Model);

subplot(2,3,4); plot(t, y(:,1,1)); title('Temp Step');
subplot(2,3,5); plot(t, y(:,1,2)); title('Rad Step');
subplot(2,3,6); plot(t, y(:,1,3)); title('Heat Step');

C = 1./SC.Complex(i).on_models{n}.params(3);
R = SC.Complex(i).on_models{n}.params(3).
/SC.Complex(i).on_models{n}.params(1) ;

plotnum = plotnum + 1;
num_onplots = num_onplots + 1;
end
end
end

for n = 1:length(SC.Complex(i).off_models)
  if ~isempty(SC.Complex(i).off_models{n})
    [y, SC.Complex(i).off_models{n}.fit, x0] = compare(SC.Complex(i).iddata_offs{n+model_extend}.ID, SC.Complex(i).off_models{n}.Model);

    if ~isempty(last_good_off_model) && preferential == 1
      [y_o, fit, x0_o] = compare(SC.Complex(i).iddata_offs{n+model_extend}.ID, last_good_off_model);
      if fit > SC.Complex(i).off_models{n}.fit
        SC.Complex(i).off_models{n}.Model = last_good_off_model;
        y = y_o;
        x0 = x0_o;
      else
        last_good_off_model = SC.Complex(i).off_models{n}.Model;
      end
    elseif g*SC.Complex(i).off_models{n}.fit > g*min_good
      last_good_off_model = SC.Complex(i).off_models{n}.
    end
  end
end

off_models{n}.Model;

end

avg_err{2}(1, i, n) = mean(y.OutputData) - mean(SC.Complex(i).iddata_offs{n+model_extend}.ID.OutputData);
max_err{2}(1, i, n) = max(y.OutputData) - max(SC.Complex(i).iddata_offs{n+model_extend}.ID.OutputData);
min_err{2}(1, i, n) = min(y.OutputData) - min(SC.Complex(i).iddata_offs{n+model_extend}.ID.OutputData);

if g*SC.Complex(i).off_models{n}.fit > g*min_good% && sum([SC.Complex(i).model_whole.params(1) < 0, SC.Complex(i).model_whole.params(3) < 0]) == 0
off_success(1, i) = off_success(1, i) + 1/length(SC.Complex(i).off_models);
fig = figure(plotnum);
set(fig,'defaultAxesColorOrder',[left_color; right_color]);
figure(plotnum);
time = SC.Weather.temp.BCIL.time(SC.Complex(i).iddata_offs{n+model_extend}.start:SC.Complex(i).iddata_offs{n+model_extend}.end);

subplot(2,3,[1,2,3]);
try
plot(time, SC.Complex(i).iddata_offs{n+model_extend}.ID.OutputData, 'k');
catch
disp('Caught an error!');
end
hold on;
plot(time, y.OutputData, 'r-');

yyaxis right;

% SC.Complex(i).iddata_offs{n}.ID.InputData(:,3));
datetickzoom

yyaxis right;
try
plot(time, SC.Complex(i).iddata_offs{n+model_extend}.RealInput);
catch
disp('Caught an error!');
end

title([Complex ', num2str(i), R = ', num2str(SC.Complex(i).off_models{n}.params(1)), C = ', num2str(SC.Complex(i).off_models{n}.params(2)), k = ', num2str(SC.Complex(i).off_models{n}.params(3)), \alpha = ', num2str(SC.Complex(i).off_models{n}.params(4))]);
legend('Measured Response (off)', ['Model
Response, ', num2str(SC.Complex(i).off_models{n}.fit), '%', 'Input (ECU Supply temp.)');

datetickzoom

[y, t] = step(SC.Complex(i).off_models{n}.model_extend).Model);

subplot(2,3,4); plot(t, y(:,1,1)); title('Temp Step');
subplot(2,3,5); plot(t, y(:,1,2)); title('Rad Step');
subplot(2,3,6); plot(t, y(:,1,3)); title('Heat Step');

C = 1./SC.Complex(i).off_models{n}.params(3); 
R = SC.Complex(i).off_models{n}.params(3).
SC.Complex(i).off_models{n}.params(1); ,

plotnum = plotnum + 1;
num_offplots = num_offplots + 1;

end

end

end

success = zeros(1, 8);
for s = 1:4
    success(1, s) = (on_success(1, s)*length(SC.Complex(i).on_models) + off_success(1, s)*length(SC.Complex(i).off_models))/(length(SC.Complex(i).on_models) + length(SC.Complex(i).off_models));
end

elseif strcmpi(eval_mode, 'concat')
    last_good_model = [];
    last_good_model_fit = [];
    success = zeros(1, 8);
    numplots = 0;
    avg_err = cell(1,8);
    max_err = avg_err;
    min_err = avg_err;

    for i = 1:4
        SC.Tent(2*i-1).params = cell(1,length(SC.Complex(i).concat_models));
        SC.Tent(2*i).params = cell(1,length(SC.Complex(i).concat_models));
        avg_err{i} = zeros(1, length(SC.Complex(i).concat_models)-2*model_extend);
        max_err{i} = avg_err{i};
        min_err{i} = avg_err{i};
    end
for n = 1:length(SC.Complex(i).concat_models) -
model_extend

io = struct('Input', [], 'Output', []);
io.Input = SC.Complex(i).iddata_concat{n}.ID.
    InputData;
io.Output = SC.Complex(i).iddata_concat{n}.ID.
    OutputData;
    opt = compareOptions('InitialCondition', io);

if ~isempty(SC.Complex(i).concat_models{n})
    [y, SC.Complex(i).concat_models{n}.fit, x0] =
        compare(SC.Complex(i).iddata_concat{n +
            model_extend}.ID, SC.Complex(i).
            concat_models{n}.Model);

    if ~isempty(last_good_model) && preferential
        == 1
        [y_o, fit, x0_o] = compare(SC.Complex(i).
            iddata_concat{n + model_extend}.ID, last_good_model);
        if fit > SC.Complex(i).concat_models{n}.
            fit
            SC.Complex(i).concat_models{n}.Model
                = last_good_model;
        y = y_o;
        x0 = x0_o;
    else
        last_good_model = SC.Complex(i).
            concat_models{n}.Model;
end

elseif SC.Complex(i).concat_models{n}.fit >
min_good

last_good_model = SC.Complex(i).
concat_models{n}.Model;

end

if i == 2 && n == 47

pause(1)

end

SC.Tent(2*i-1).params{n}(1) = SC.Complex(i).
concat_models{n}.params(1);

SC.Tent(2*i-1).params{n}(2) = SC.Complex(i).
concat_models{n}.params(2);

SC.Tent(2*i-1).params{n}(3) = SC.Complex(i).
concat_models{n}.params(3);

SC.Tent(2*i-1).params{n}(4) = SC.Complex(i).
concat_models{n}.params(5);

SC.Tent(2*i-1).params{n}(5) = SC.Complex(i).
concat_models{n}.params(6);

SC.Tent(2*i).params{n}(1) = SC.Complex(i).
concat_models{n}.params(7);

SC.Tent(2*i).params{n}(2) = SC.Complex(i).
concat_models{n}.params(8);

SC.Tent(2*i).params{n}(3) = SC.Complex(i).
concat_models{n}.params(4);

SC.Tent(2*i).params{n}(4) = SC.Complex(i).
concat_models{n}.params(9);

SC.Tent(2*i).params{n}(5) = SC.Complex(i).
concat\_models\{n\}.params(6);

avg\_err\{i\}(1, n) = mean(mean(y.OutputData) -
               mean(SC.Complex(i).iddata\_concat\{n+
                model\_extend}.ID.OutputData));

max\_err\{i\}(1, n) = max(max(y.OutputData) -
                     max(SC.Complex(i).iddata\_concat\{n+
                      model\_extend}.ID.OutputData));

min\_err\{i\}(1, n) = max(min(y.OutputData) -
                    min(SC.Complex(i).iddata\_concat\{n+
                     model\_extend}.ID.OutputData));

if abs(avg\_err\{i\}(1,n)) < 1/1.8 && abs(
    max\_err\{i\}(1,n)) < 2/1.8 && abs(min\_err\{i\}(1,n)) < 2/1.8

success(1, i) = success(1, i) + 1/length(
    SC.Complex(i).concat\_models);

end

tit = {'Model Fit to Current Dataset', '
    Model (prev. ~45 minutes) Prediction of
    System Performance'};

if plotnum < 50
    fig = figure(plotnum);
    figure(plotnum);
    plotnum = plotnum + 1;
    time = SC.Weather.temp.BCIL.time(SC.
        Complex(i).iddata\_concat\{n+
        model\_extend}.start:SC.Complex(i).
try
  plot(time, SC.Complex(i).
    iddata_concat{n+modelextend}.ID.
    OutputData, 'k', time, y.
    OutputData, 'r-');
  catch
    pass
  end

  title({titl+model_extend}, ['Coupled
    Tents ', num2str(2*i-1), ' & ',
    num2str(2*i), ', Known Inputs '])
  datetickzoom
  else
    disp('Warning! Truncated plots to
      prevent system crash.');
  end
end
end
end
end
end

SC_n = SC;
B.4.5  grey_master.m

grey_master.m is the driver for all of the functions described above. It is where the user selects what test data to use for model fitting, how the data is concatenated before conducting the model fit, and how it is displayed after conducting the model fit.

```matlab
1 clear all;
2 clf;
3 close all;
4
5 %% Test Date
6 testdate = 'MAY 22 2017';
7
8 %% Other parameters
9 est = 0;  %1 to estimate parameters, 0 to fix parameters at initial guesses
10
11 inp_del = 0;  %heater input delay
12 uncontrolled = 1:4;
13 controlled = 5:8;
14 grouping = 'whole';  %'concat' to concatenate datasets before fitting, 'individual' to not do so.
15 group_degree = 1;  %1 concatenates datasets into groups of ONE on-off set. 2 concatenates into TWO such sets, etc ...
16 min_qual = -inf;  %variable in range (-inf, 100), indicates minimum fit quality that will be displayed.
17 min_length = 0;  %minimum length for which a fit will be attempted on a dataset
18 extension = 0;  %number of datasets forward to apply
```

315
each model

mass_flow_est = .015; %estimated constant to stand in for
mass flow rate of input air

buffer = 0; %1 to take the middle section (buffer
# samples away from endpoints) of turn-on/turn-off events
for fitting, 0 to use the entire event.

pad = 1; %1 to pad the beginning of datasets
with initial rest conditions, 0 to not

neg = 1; %1 to show good fits, -1 to show bad
fits

preferential = 0; %1 to favor previous good fits over
current bad fits, 0 to always use current fit

%%% Collect data to be fit

home = pwd;

cd('.../.../../Compiled Data');

load([testdate, ' Post-Processing.mat']);

cd('../2017 Spring')

for i = 1:length(uncontrolled)
    [SC_u.Tent(uncontrolled(i)), ~, ~] = detect_heat_u(SC_u.
        Tent(uncontrolled(i)));
end

cd(home);

t_start = '22-MAY-2017 16:00:00'; %DD-MON-YYYY HH:MM:SS

t_end = -1;

% t_start = '23-MAY-2017 21:00:00'; %DD-MON-YYYY HH:MM:SS
t_end = '23-MAY-2017 04:00:00';
```matlab
fit_on = cell(8, length(inp_del));
fit_off = cell(8, length(inp_del));
fit_who = cell(8, length(inp_del));

SC_n = grey_package(SC_u, t_start, t_end, 'mode', buffer, 'static', pad);

if strcmp(grouping, 'concat')
%    SC_n = grey_concat(SC_n, group_degree, pad);
    SC_n = grey_concat_2(SC_n, group_degree, pad);
end

try
    SC_n = grey_fit(SC_n, min_length, inp_del, grouping,
                    mass_flow_est, pad, est);
catch
    0;
end
clf;
close all;

[SC_n, success, avg_err, max_err, min_err, abs_err] =
    grey_eval(SC_n, grouping, min_qual, neg, extension, preferential);

if strcmpi(grouping, 'concat') || strcmpi(grouping, 'whole')
    avg_err_mean = zeros(1, 8);
    avg_err_std = avg_err_mean;
    max_err_mean = avg_err_mean;
    max_err_std = avg_err_mean;
    min_err_mean = avg_err_mean;
```
min_err_std = avg_err_mean;

for i = 1:8
    avg_err_mean(1, i) = mean(avg_err{1}(1, i, :));
    avg_err_std(1, i) = std(avg_err{1}(1, i, :));
    avg_err_max(1, i) = max(abs(avg_err{1}(1, i, :)));

    max_err_mean(1, i) = mean(max_err{1}(1, i, :));
    max_err_std(1, i) = std(max_err{1}(1, i, :));
    max_err_max(1, i) = max(abs(max_err{1}(1, i, :)));

    min_err_mean(1, i) = mean(min_err{1}(1, i, :));
    min_err_std(1, i) = std(min_err{1}(1, i, :));
    min_err_max(1, i) = min(abs(min_err{1}(1, i, :)));
end

else
    avg_err_mean_on = zeros(1, 8); avg_err_mean_off = zeros(1, 8);
    avg_err_std_on = avg_err_mean_on; avg_err_std_off = avg_err_mean_off;
    max_err_mean_on = avg_err_mean_on; max_err_mean_off = avg_err_mean_off;
    max_err_std_on = avg_err_mean_on; max_err_std_off = avg_err_mean_off;
    min_err_mean_on = avg_err_mean_on; min_err_mean_off = avg_err_mean_off;
    min_err_std_on = avg_err_mean_on; min_err_std_off = avg_err_mean_off;
end
for i = 1:8
    avg_err_mean_on(1, i) = mean(avg_err{1}(1, i, :));
    avg_err_mean_off(1, i) = mean(avg_err{1}(1, i, :));
    avg_err_std_on(1, i) = std(avg_err{1}(1, i, :));
    avg_err_std_off(1, i) = std(avg_err{1}(1, i, :));

    max_err_mean_on(1, i) = mean(max_err{1}(1, i, :));
    max_err_mean_off(1, i) = mean(max_err{1}(1, i, :));
    max_err_std_on(1, i) = std(max_err{1}(1, i, :));
    max_err_std_off(1, i) = std(max_err{1}(1, i, :));

    min_err_mean_on(1, i) = mean(min_err{1}(1, i, :));
    min_err_mean_off(1, i) = mean(min_err{1}(1, i, :));
    min_err_std_on(1, i) = std(min_err{1}(1, i, :));
    min_err_std_off(1, i) = std(min_err{1}(1, i, :));
end
end

bin = linspace(0, 4, 20);
numa = zeros(1, length(bin)-1);
umax = numa;
umin = numa;
umabs = numa;
for i = 2:length(bin)
numa(1, i-1) = sum(sum(abs(avg_err{1}) < bin(i)/1.8 &
abs(avg_err{1}) > bin(i-1)/1.8));
numax(1, i-1) = sum(sum(abs(max_err{1}) < bin(i)/1.8 &
abs(max_err{1}) > bin(i-1)/1.8));
numin(1, i-1) = sum(sum(abs(min_err{1}) < bin(i)/1.8 &
abs(min_err{1}) > bin(i-1)/1.8));
umabs(1, i-1) = sum(sum(abs(abs_err{1}) < bin(i)/1.8 &
abs(abs_err{1}) > bin(i-1)/1.8));
end

figure();
bar(bin(2:end), numa);
xlim([0, 10]);
ylabel(' # of predictions');
xlabel('Absolute error in predicted average temperature (deg. F)');
title('Model Average Temp. Prediction');

figure();
bar(bin(2:end), numax);
xlim([0, 10]);
ylabel(' # of predictions');
xlabel('Absolute error in predicted max temperature (deg. F)');
title('Model Max. Temp. Prediction');

figure();
bar(bin(2:end), numin);
xlim([0, 10]);
ylabel(' # of predictions');
B.5 MATLAB FOB simulation

The MATLAB FOB simulation suite is an object-oriented environment in which various load, generator, tent, weather, and control models can interact with one another. It has the following major components:

1. duplex.m
2. ecu.m
3. gen.m
4. identifier.m
5. latrine_pump.m
6. lc.m
7. mc.m
8. weather.m
9. camp.m
10. test_sim.m

Items 1-8 are modules that represent the dynamics, control logic, and physical properties of FOB subsystems. Item 9, camp.m, is a framework object that ties 1-8 together and sets the terms of their interactions. Item 10, test_sim.m, allows for the rapid specification, execution, and logging of simulation tests.

B.5.1 duplex.m

The duplex module represents the thermal dynamics of a tent complex at the BCIL. An instance of the duplex object can have arbitrary model parameters, but the model topology is fixed as that introduced in 2-5. A different module should be made if other model topologies are to be analyzed.

In the example shown here, the duplex module has statically-defined parameters. There is no reason that the parameters could not be dynamically specified with an additional argument to the constructor method [e.g. duplex = duplex(ecu1, ecu2, weather, params)].

classdef duplex < handle
    %DUPLEX Summary of this class goes here
    % Detailed explanation goes here

    properties
        %Connected objects
            ecu1
            ecu2
            Weather

        %System parameters
            Rte1   %Duplex tent 1, thermal resistance to environment
            Rte2   %Duplex tent 2, thermal resistance to
environment

C1    %Duplex tent 1, lumped thermal capacitance
C2    %Duplex tent 2, lumped thermal capacitance
k1    %Duplex tent 1, solar absorption coefficient
k2    %Duplex tent 2, solar absorption coefficient
Rtt   %Duplex tent-to-tent thermal resistance
flow_est %Duplex heaters, time constant estimate

%System states
T1
T2
Q1
Q2
state

%Inputs
temp
rad
end

properties (Dependent)
%Define tent system in state space format
A
B
end

methods
%Constructor method
function duplex = duplex(ecu1, ecu2, weather)
    duplex.ecu1 = ecu1;
    duplex.ecu2 = ecu2;
    duplex.Weather = weather;
    duplex.Rte1 = 0.0016;
    duplex.Rte2 = 0.0020;
    duplex.C1 = 474000;
    duplex.C2 = 461000;
    duplex.k1 = 8.65;
    duplex.k2 = 4.53;
    duplex.Rtt = 0.0039;
    duplex.flow_est = 0.0258;
    duplex.state = init_state;
end

% Getter methods
function A = get.A(self)
    A = \[-(1./self.C1)*(self.Rtt+self.Rte1)/(self.Rtt*
    C1), 0; ...
         0,
         -self.flow_est, 0,
         0; ...
    1/(self.Rtt*self.C2), 0,
    ; ...

324
0,

0, 0,

-s .flow_est];
end

function B = get_B(self)
B = [1./(self.Rte1* self.C1), self.k1/self.C1, 0,
0;...
0, 0,
self.flow_est, 0;...
1./(self.Rte2* self.C2), self.k2/self.C2, 0,
0;...
0, 0, 0,
self.flow_est];  %
[1./(par(1)*par(2)), par(3)/par(2)
, 1/par(2)];  % [1./(par(1)*par(2)),
par(3)/par(2), aux(1)/par(2)];
end

function set_inp(self, temp, rad)
self.temp = temp;
self.rad = rad;
end

function [slopes] = tentsim(self, t, statev)
self.T1 = statev(1);
self.Q1 = statev(2);
self.T2 = statev(3);
self.Q2 = statev(4);

self.state = [self.T1, self.Q1, self.T2, self.Q2
I; infs = [self.temp, self.rad, self.ecu1.heat_out, self.ecu2.heat_out];

try
    slopes = self.A*self.state' + self.B*inps';
catch
    pause(0);
end
end
end

B.5.2 ecu.m

The ecu module is very minimal: it has a resistance property that it presents to the generator module and a heat output property that it presents to the duplex module, both based on the mode command of the lc (local controller) module. A camp simulation typically involves several instances of the ecu module, where each instance is assigned a unique instance of the lc module.

classdef ecu < handle
    %ECU Summary of this class goes here
    % Detailed explanation goes here

    properties
        LC
        Gen
        heater_res = 4.8; %Per-phase heater resistance

end
vent_res = 43.2;

end

properties (Dependent = true)
  RL
  heat_out
end

methods
  %Constructor method
  function ecu = ecu(LC)
    ecu.LC = LC;
    ecu.Gen = Gen;
  end

  %Getter methods
  function value = get.Rl(self)
    if (self.LC.heat_state || self.LC.vent_state)
      value = 1/(self.LC.heat_state/self.heater_res
               + self.LC.vent_state/self.vent_res);
    else
      value = 1e9;
    end
  end

  function heat_out = get.heat_out(self)
    if self.Gen.ready()
      heat_out = 3*120.^2/self.Rl - 3*120.^2/self.
                  vent_res;
    else

B.5.3 gen.m

The gen (generator) module represents the steady-state fuel consumption characteristics, basic electromechanical dynamics and regulation properties, and automatic dispatch logic of a parallel bank of diesel generators. It has a property for each voltage stream (see Section 4.3) that the identifier module measures, which it determines either based on its own state (such as for frequency) or based on a behavioral model of that stream based on measurements at the BCIL (such as for seventh harmonic voltage).

```matlab
classdef gen < handle
    %GEN Summary of this class goes here
    % Detailed explanation goes here
    properties (Access = protected)
        ID = 'gen';
        % Connected objects
        ECU
        latrine_pump
        % Machine parameters
        P = 2; % Number of poles (*not* pole pairs)
```
\[ rs = 0.1; \quad \text{Stator winding resistance} \]
\[ L_{ls} = 0.00011; \quad \text{???} \]
\[ L_{mq} = 0.003; \quad \text{???} \]
\[ L_{md} = 0.003; \quad \text{???} \]
\[ J = 0.1; \quad \text{Rotor inertia} \]
\[ M = 0.0827; \quad \text{Stator-field winding mutual inductance} \]
\[ M = 0.0227; \]
\[ R_f = 5; \quad \text{Field winding resistance} \]
\[ L_f = 3; \quad \text{Field winding self-inductance} \]
\[ V_{ff} = 20; \quad \text{Field winding applied voltage} \]
\[ v_{ref\_max} = 170; \quad \text{Maximum instantaneous phase-neutral stator voltage} \]
\[ \text{max\_droop} = 2; \quad \text{Generator voltage drop at max load current} \]
\[ w_{ref} = 377; \quad \text{Nominal electrical frequency (rad/s)} \]
\[ K_f = 1; \quad \text{Field winding controller gain} \]
\[ K_m = 2; \quad \text{Mechanical prime mover controller gain} \]
\[ I_{cap} = 167; \quad \text{Per-phase current capacity of a single generator (RMS)} \]
\[ \text{gen\_ready} = \text{false}; \]
\[ R_o = 0.009; \]

\[ \%\% \text{ Machine control properties} \]
\[ \text{on\_timer} = 10; \]
\[ \text{off\_timer} = 300; \]

\[ \%\% \text{ Machine States} \]
qs = 0; % stator current (DQ0 transform)
ids = 0; % D stator current (DQ0 transform)
wr = 0; % Electrical rotor frequency (rad/s)
th = 0; % Electrical rotor position (rad)
iff = 0; % Field current (A)
sig = 0; % Accumulator for PI control of field
        winding and mech. speed control
sig2 = 0; % 

sig3 = 0; % Timer for generator on transitions
sig4 = 0; % Timer for generator off transitions
turning_on = false;
turning_off = false;
end

properties
    Rbase = 40;
    state = [0 0 0 0 0 0 0];
    N = 2; % # active generators... Approximates
           paralleling behaviors
    N_last = 2;
    N_next = 2;

    auto_dispatch = true;
    force_on = false;
    force_off = false;
    base_load = 40; % Per-phase resistance of base
          load
end

properties (Dependent = true)
Lq
Ld
I_max
I_min
R
f
S1
S3
P7

end

methods

%Class constructor
function self = gen(ECU, latrine_pump)
    self.ECU = ECU;
    self.latrine_pump = latrine_pump;
end

%Getter functions
function value = get.Lq(self)
    value = (self.Lls+self.Lmq)/self.N;
end
function value = get.Ld(self)
    value = (self.Lls+self.Lmd)/self.N;
end
function value = get.I_max(self)
    value = self.I_cap*self.N*0.8;
end
function value = get.I_min(self)
    value = self.I_cap*(self.N-1)*0.6;
function value = get.Rl(self)
    resistances = zeros(1,length(self.ECU));
    for i = 1:length(self.ECU)
        resistances(i) = self.ECU{i}.Rl;
    end

    value = 1/sum([1./resistances, 1/self.latrine_pump.Rl, 1/self.base_load]);
end

function value = get.f(self)
    value = self.wr/2/pi + 0.05*randn(1);
end

function value = get.S3(self)
    genbase = 1.5 - (self.N_next - 1)*0.5;
    ecubase = 0;
    for i = 1:length(self.ECU)
        ecubase = ecubase - 0.1*self.ECU{i}.LC.heatstate/(self.N^2);
    end

    value = genbase + ecubase + 0.05*randn(1);
end

function value = get.S1(self)
    value = 170*self.Rl/(self.Rl + self.Ro/self.N) + 0.35*randn(1);
end

function value = get.P7(self)
    load = 100 * 120 / self.Rl / self.I_cap / self.N_next;
    value = sum([0.5396 - 0.0124*load ...]
.0017*load^2, ...
-2.2924e-5*load^3, ...
9.8223e-8*load^4, ...
0*(self.N_next == 1), ...
-0.663*(self.N_next == 2)) + .05*randn(1);
end

%Setter functions
function set.force_on(self, val)
    try
        if strcmpi(val{2}, 'mc') || strcmpi(val{2}, 'gen')
            self.force_on = val{1};
        end
        catch
            disp('Generator override failed!');
    end
end

function set.force_off(self, val)
    try
        if strcmpi(val{2}, 'mc') || strcmpi(val{2}, 'gen')
            self.force_off = val{1};
        end
        catch
            disp('Generator override failed!');
    end
end

function set.auto_dispatch(self, val)
    try
if strcmpi(val{2}, 'mc') || strcmpi(val{2}, 'gen')
    self.auto_dispatch = val{1};
end

catch
    disp('Generator override failed!')
end
end
function set_base_load(self, val)
    try
        if strcmpi(val{2}, 'camp')
            self.base_load = val{1};
        end
    catch
        disp('Generator base load manipulation failed!')
    end
end

%Simulation functions
function value = ready(self)
    value = self.gen_ready;
end
function [slopes] = gensim(self, t, statev)
    % disp(num2str(t));
    % % Update generator states
    self.iqs = statev{1};
    self.ids = statev{2};
self.wr  = statev(3);
self.th  = statev(4);
self.iff = statev(5);
self.sig  = statev(6);
self.sig2 = statev(7);

%%% Indicate generator ready if initial turn-on transient has passed
if t > 1
  self.gen_ready = 1;
end

%%% Update generator load

%%% Compute rotor-angle-dependent machine parameters
lqs = self.Lq.*self.iqs;  % Q axis stator flux
lds = self.Ld.*self.ids+self.M.*self.iff;  % D axis stator flux

%%% Compute rotor/stator voltages/currents
vqs = (self.iqs).*(self.Rl+self.Ro/self.N);  % Voltage drop across Q axis stator resistance
vds = (self.ids).*(self.Rl+self.Ro/self.N);  % Voltage drop across D axis stator resistance
vtot = sqrt(vqs*vqs + vds*vds);  % Total voltage drop
across stator resistance

\[ \text{itot} = \sqrt{\text{self.iqs}^2 \cdot \text{self.iqs}^2 + \text{self.ids}^2 \cdot \text{self.ids}^2} \]; \quad \% \text{Total stator current} \\

\[ \text{vref} = \text{self.vref\_max} - \text{self.max\_droop} \cdot (\text{itot} / (\text{self.I\_cap} \cdot \text{self.N})) \]; \quad \% \text{Reference stator voltage} \\

\[ \text{Vf} = \text{self.Kf} \cdot (\text{vref} - \text{vtot}) + 10 \cdot \text{self.sig}; \] \\

\% \text{Voltage supplied to field winding} \\

\[ \text{Tp_m} = -\text{self.N} \cdot \text{self.Km} \cdot (\text{self.wref} - \text{self.wr}) - 20 \cdot \text{self.sig2}; \] \quad \% \text{Prime mover (applied) torque} \\

\text{if \ sum(isnan([\text{lqs, lds, vqs, vds, vtot, itot, vref, Vf, Tpm}])) > 0} \\
\text{pause(0);} \\
\text{end} \\

\%\% \text{Auto-dispatch logic} \\
\text{if \sim \text{self.turning\_on}} \\
\text{if \ (\text{itot} / \sqrt{2}) > \text{self.I\_max} \&\& \text{self.auto\_dispatch}) || (\text{self.force\_on} \&\& \sim \text{self.auto\_dispatch})} \\
\text{if \text{self.sig3} > 0} \\
\text{if \ t - \text{self.sig3} > \text{self.on\_timer}} \\
\text{\quad \text{self.turning\_on} = \text{true};} \\
\text{\quad \text{self.N\_last} = \text{self.N};} \\
\text{\quad \text{self.N\_next} = \text{self.N} + 1;} \\
\text{\quad \text{self.sig3} = 0;} \\
\text{\quad end} \\
\text{else} \\
\text{\quad \text{self.sig3} = \text{t};}
end
else
    self.sig3 = 0;
    self.turning_on = false;
end
else
%MANAGE GENERATOR TURN-ON
    if self.sig3 > 0
        if t - self.sig3 < 0.5
            self.N = self.N_last + floor(1e3*abs(t - self.sig3))/(5e2);
        else
            self.N = ceil(self.N_next);
            self.N_last = self.N;
            self.turning_on = false;
            self.force_on = {false, self.ID};
        end
    else
        self.sig3 = t;
    end
end

% Turn-off logic
if ~self.turning_off
    if (itot/sqrt(2) < self.I_min && self.autodispatch) || (self.force_off && ~self.autodispatch)
        if self.sig4 > 0
            if t - self.sig4 > self.off_timer
                self.turning_off = true;
            end
        end
    end
end
self.N_last = self.N;
self.N_next = self.N - 1;
self.sig4 = 0;

end

else
    self.sig4 = t;
end

else
    self.sig4 = 0;
    self.turning_off = false;
end

else

% MANAGE GENERATOR TURN-OFF

    if self.sig4 > 0
        if t - self.sig4 < 0.1
            self.N = self.N_last - floor(1e4*abs(t-self.sig4))/(1e3);
        else
            self.N = floor(self.N_next);
            self.turning_off = false;
            self.force_off = {false, self.ID};
        end
    else
        self.sig4 = t;
    end

end

%% Compute state derivatives

dsl = (1/self.Lq)*(-vqs - self.rs*self.iqs/self.N - self.wr*lds); % dIq/dt
\[ ds2 = \left(\frac{1}{self.Ld}\right) \left(-vds - self.rs*\frac{self.ids}{self.N} + self.wr*\frac{self.Lq*self.iqs}{self.N}\right); \text{ \%} \frac{dId}{dt} \]

\[ T = \left(\frac{3}{2}\right) \left(\frac{self.P}{2}\right) \left(\frac{lds*self.iqs - lqs*\frac{self.ids}{self.N}}{self.N*\frac{self.J}{self.N}}\right); \text{ \%} \text{cross}(J,B) \]

\[ ds3 = -\left(\frac{self.P}{2}\right) \left(Tpm - T\right) / (self.N*\frac{self.J}{self.N}); \text{ \%} \text{Inertia scaled by number of generators} \]

\[ ds4 = self.wr; \text{ \%} \text{Rotor angular velocity} \]

\[ ds5 = (Vf/\text{self.Lf}) - (\text{self.Rf}/\text{self.Lf}) * self.iff; \text{ \%} \text{Rate change of field current} \]

\[ ds6 = vref-vtot; \text{ \%} \text{Integrate stator voltage error} \]

\[ ds7 = (\text{self.wref}-\text{self.wr}*\frac{self.P}{2})*\frac{self.N}{self.P}; \text{ \%} \text{Integrate rotor electrical frequency error} \]

if \( \text{sum(isnan([ds1 ds2 ds3 ds4 ds5 ds6 ds7])}) > 0 \)
\[
\text{pause}(0); \text{ end} 
\]

slopes = \[ ds1 ds2 ds3 ds4 ds5 ds6 ds7 \]^\prime; 
\[
\% \text{slopes} = \left[0 0 0 0 0 0 0\right]^\prime; 
\]

end

end
B.5.4 identifier.m

The identifier module functions as described in Section 4.4. Each local controller instance in the simulation has its own unique instance of the identifier object, from which it receives an estimate of the number of ECUs operating at every moment and flags to indicate that an identification or collision-handling routine is in progress.

% TO DO: Fix load estimation and assignment of transients to exemplar load
% indices. REDUCE medfilt1 window size!

classdef identifier < handle
    %The identifier identifies load events by their corresponding line
    % voltage transients.
    % Detailed explanation goes here

    properties (Access = private)
        % Processing options
        medsize
        stream = {'f', 'P7'};
        class = {'ECU on', 'ECU off', 'Gen. on', 'Gen. off'};
        integration_tolerance = struct('f', .15, 'P7', .15);
        overhead = 5;

        % Class 1: ECU on,
        % Class 2: ECU off, Class 3:
        % Class 4: Gen. off

340
match_threshold = struct('f', [0.3, 0.3, 0.3, 0.3], ...
                     'P7',[0.3, 0.3, 0.3, 0.3]);

end

properties

idnum

% Processor variables

Wlen = 5; % Default duration of analysis window (seconds)

W = 1001; % Default length of analysis window (samples)

identifying = 0;
sensed = 0;
wait % # samples more to read before processing

ecu_class = [1 0 2];
gen_class = [3 0 4];
ind = 0;
inds
series
logging_exemplars = 0;
load_increment = 5;
exemplars = {};

341
\begin{verbatim}
tstep
lookfor = 0;
collision = 0;
actionpause = 1;
collisioncheck = 0;
gencheck = 0;
P7_prev = -1;
P7_after = -1;
init_wait = 5;
freq_base = [];

trans_baseload
trans_baseload_rel
baseload

% # of ECUs heating, # of generators active
num_ecu = [0 0]; [%current, previous]
num_gen = [1 1]; [%current, previous]

% ECU characteristics
H_PWR = 9; %CHANGED 07FEB18 TO FACILITATE TESTING—
       REAL VALUE = 9; %Heater power (kW)
V_PWR = 1; %Vent power (kW)
tot_ecu;

% Generator bank characteristics
C = 60; %Single generator capacity (kW)
maxload = 0.8; %Load threshold for generator turn-on

% Transient ID method
\end{verbatim}
method = 'god';

%Connected objects
MC
Gen
end

properties (Dependent)
adv
adv_search
analysis_loc
analysis_loc_x
estload
load_rel
is_overload
will_overload
room_for_one
end

methods

%Constructor
function identifier = identifier(tot_ecu, idnum)
    identifier.tot_ecu = tot_ecu;
    identifier.idnum = idnum;
end

function delete(self)
end

%Property getters
function adv = get.adv(self)
    adv = ceil(self.analysis_loc+6/self.tstep);
    adv = self.analysis_loc + floor(1.5/self.tstep);
    adv = floor(0.05/self.tstep);
end

function adv_search = get.adv_search(self)
    adv_search = floor(0.1/self.tstep);
end

function analysis_loc = get.analysis_loc(self)
    analysis_loc = 200; % floor (self.W/3);
end

function estload = get.estload(self)
    if strcmpi(self.method, 'god')
        estload = self.MC.load;
    elseif strcmpi(self.method, 'mortal')
        estload = self.num_ecu(1)*(self.H_PWR) + self.tot_ecu*self.V_PWR;
    elseif strcmpi(self.method, 'test')
        estload = self.num_ecu(1)*(self.H_PWR) + self.tot_ecu*self.V_PWR + self.baseload;
    end
end

function is_overload = get.is_overload(self)
    if self.estload > self.C*self.num_gen(1)*self.maxload - self.overhead
        is_overload = 1;
    else
        is_overload = 0;
    end
end
function will_overload = get.will_overload(self)
        num_gen(1) - self.overhead*1.2
            will_overload = 1;
        else
            will_overload = 0;
    end
end

function load_rel = get.load_rel(self)
    load_rel = 100*self.estload/(self.C*self.num_gen
(1));
end

function trans_baseload_rel = get.
    trans_baseload_rel(self)
    trans_baseload_rel = 100*self.trans_baseload/
        (self.C*self.num_gen(1));
end

function room_for_one = get.room_for_one(self)
    %Checking if you can run under ONE gen
        H_PWR
        room_for_one = 1;
    else
        room_for_one = 0;
    end
end
function ID = ID(self)
    ID = 0;

%Add latest frequency datapoint
for i = 1:length(self.stream)
    self.series.(self.stream{i}) = circshift
        (self.series.(self.stream{i}), -1);
    self.series.(self.stream{i})(end) = self.Gen
        .(self.stream{i});
end

if self.init_wait == -1
    %GOD MODE
    if strcmpi(self.method, 'god')
        self.num_ecu = circshift(self.num_ecu,
            [0,-1]); self.num_ecu(end) = self.MC.
            num_ecu;
        self.num_gen = circshift(self.num_gen,
            [0,-1]); self.num_gen(end) = self.MC.
            num_gen;
        self.sensed = circshift(self.sensed,
            [0,-1]); self.sensed(end) = (self.
            num_ecu(end) ~= self.num_ecu(end-1))
            || (self.num_gen(end) ~= self.num_gen(
                end-1));

        if self.sensed(1) == 1
            self.identifying = 1;
        end
    end
end
if self.sensed(end) == 1
    self.num_ecu(1) + self.V_PWR* self.
    tot_ecu;
end

%disp(self.identify);%

%Use a-priori knowledge of ecu/gen. transients
if self.num_ecu(1) == self.num_ecu(2)
    ID = self.ecu_class( 2 + sign( self.
    num_ecu(1)-self.num_ecu(2) ) );
    if self.logging_exemplars
        self.exemplar_update(self.
        trans_base_load_rel, ID);
    end
self.identifying = 0;
elseif self.num_gen(1) == self.num_gen(2)
    ID = self.gen_class( 2 + sign( self.
    num_gen(1)-self.num_gen(2) ) );
    if self.logging_exemplars
        self.exemplar_update(self.
        trans_base_load_rel, ID);
    end
self.identifying = 0;
else
    ID = 0;

end

else

%Discard a-priori knowledge of ecu/gen.
transients
clear num_ecu; clear num_gen;

%Is it time for window analysis?
if self.wait == 0

%Check if a generator was party to the collision
if self.gencheck
    if ismember('P7', self.stream)
        self.P7_after = mean(self.series.P7);
        trans = self.P7_after - self.P7_prev;

%Exemplar Selection
xmplrs = self.get_exemplars(
    self.load_rel, self.num_gen(1));

score = 10*[self.match_threshold.P7(3);
    self.match_threshold.P7(4)];
match_thresh = score/10 *(1 + 0.5*(self.num_gen(1)-1)) ;
for i = 3:4
    %Generate match scores
    for each exemplar in each stream
    %Initialize class score vector
    if ~isempty(xmplrs{1,i}.P7)
        sub_score = abs(trans -xmplrs{1,i}.P7{1})/abs((1.5* xmplrs{1,i}.P7{1})
    score(i-2) = min([ sub_score; 10* match_thresh(i-2) ]); 
    end
end

%Find event classes that match transients across ALL streams
match_ind = find(score <= match_thresh);
% match_ind = find(mean(score,1) == self.match_threshold);
match_candidates = score(match_ind);
thresh_candidates =
match_thresh(match_ind);

if ~isempty(match_ind)
    [~, best_ind] = max((thresh_candidates - match_candidates)./thresh_candidates);
    genadj = [1, -1];
    self.num_gen = [self.num_gen(1)+genadj(match_ind(best_ind)), self.num_gen(1)];

    disp('collision contained a generator transient!');
else
    disp('collision contained no generator transient!');
end

else
    disp('not equipped to determine if collision involved a generator!');
end

self.gencheck = 0;
elseif self.collisioncheck
    figure(1);
    plot(self.series.f);
    trans = self.series.f - self.
```python
find_base(self.series.f);
[st, -] = self.is_trans_f(trans,
0.2, 5, 'narrow');

if st%max(self.series.f(1:floor(
self.W/2)) - self.find_base(
self.series.f(1:floor(self.W
/2)))) > 0.2 || min(self.
series.f(floor(self.W/2):end)
- self.find_base(self.series.f
(floor(self.W/2):end))) < -0.2
    disp('Collision detected! (Observer)');
    self.wait = self.W;
    self.actionpause = self.W;
else
    disp('No collision detected
! Checking immediately for a missed generator transient');
    self.wait = 1;
    self.actionpause = self.W;
end
self.collisioncheck = 0;
self.gencheck = 1;
%Search frequency series for a
transient
else

    if self.ind <= 0
        [self.inds, adv_to] = self.
trans_search(self.series,
```

351
if self.inds.P7 > 0 && self.inds.f > 0
    self.ind = min([self.inds.f, self.inds.P7]);
    adv_to = min([adv_to, 
                   self.inds.P7+floor(1/self.tstep)]);
elseif self.inds.f > 0
    self.ind = self.inds.f;
    self.inds.P7 = self.inds.f;
elseif self.inds.P7 > 0
    self.ind = self.inds.P7;
    self.inds.f = self.inds.P7;
    adv_to = self.inds.P7+floor(1/self.tstep);
end
end
%Does the window contain a transient?
if self.ind > 0
    self.freq_base = self.find_base(self.series.f);
    figure(1);
    plot(self.series.f);
    %Indicate that a transient is
being analyzed
self.identifying = 1;

disp(['After Trans Search:
    i_f = ', num2str(self.inds.f), ', i_7 = ', num2str(self.inds.P7)]);

%Is the transient in the desired position?
if self.ind <= self.analysis_loc
    figure(1);
    plot(self.series.f);
    %Indicate that a transient is being analyzed
    self.identifying = 1;
    %Identify the transient
    for i = 1:length(self.stream)
        %
        transient.(self.stream{i}) = self.series.(self.stream{i})(1:self.analysis_loc + 2/self.tstep);
        transient.(self.stream{i}) = self.series.(self.stream{i})(1:end);
    end

    [self.inds, adv_to] =
self.trans_search (self .series, []);

if self.inds.P7 > 0 &&
self.inds.f > 0
    self.ind = min([self.
        inds.f, self.inds.
        P7]);
    adv_to = min([adv_to,
        self.inds.P7 +
        floor(1/self.tstep
    )]);
elseif self.inds.f > 0
    self.ind = self.inds.
        f;
    self.inds.P7 = self.
        inds.f;
elseif self.inds.P7 > 0
    self.ind = self.inds.
        P7;
    self.inds.f = self.
        inds.P7;
    adv_to = self.inds.P7
        +floor(1/self.
        tstep);
end

%adv_to is how many
samples you advance
%through without further
%identifier doesn't find anything this time... This can happen if trans_search
%sees something that's just too far out in %the analysis window this time, and we
%don't want to leapfrog it. That in mind,
%make sure adv_to stays within a second or %two of the analysis location.
adv_to = min([adv_to, self.analysis_loc + floor(2/self.tstep)]);

[ID, ~, true_nd] = self.
  trans_class(self.
    series, self.inds);

if self.lookfor > 0
  if ID == 1 && self.
    lookfor == 1
  self.num_ecu = [ self.num_ecu
    (1)+1, self.
    num_ecu(1)];
self.wait =
    true_nd;
self.actionpause = true_nd;
elseif ID == 2 &&
    self.lookfor == 2
    self.num_ecu = [
        self.num_ecu
        (1)-1, self.
        num_ecu(1)];
self.wait =
    true_nd;
self.actionpause = true_nd;
else
    %Collision! (Participant)
    %Measure pre-
    collision P7
    level
    if ismember('P7',
        self.stream)
        self.P7_prev
        = mean(
            self.
            series.P7
            (1:self.
            analysis_loc
            - 0.1/
self.tstep
);
else
disp('not equipped to test P7 for Generator transient! Must add P7 to input streams');
end

%Indicate that collision has occurred
disp('Collision detected! (Participant)');

self.collision = 1;
self.gencheck = 1;
self.wait = 2*self.W;
self.actionpause = 2*self.W;
end

self.lookfor = 0;

%If the unit did NOT just create a %transient... see if the transient of %another can be
identified
else
  if ID == 1
    self.num_ecu = [
      self.num_ecu
      (1)+1, self.
      num_ecu(1)];
    self.wait =
      true_nd;
    self.actionpause
      = true_nd;
  elseif ID == 2
    self.num_ecu = [
      self.num_ecu
      (1)-1, self.
      num_ecu(1)];
    self.wait =
      true_nd;
    self.actionpause
      = true_nd;
  elseif ID == 3
    self.num_gen = [
      self.num_gen
      (1)+1, self.
      num_gen(1)];
    self.wait =
      true_nd;
    self.actionpause
      = true_nd;
  elseif ID == 4

358
self.num_gen = [
    self.num_gen
(1)-1, self.
num_gen(1)];
self.wait =
trueND;
self.actionpause = trueND;
else

%Measure pre-
collision P7
level
if ismember('P7',
self.stream)
self.P7_prev = mean(
    self.
series.P7
(1:self.
analysis_loc
- 0.1/
self.tstep
));
else
disp('not
equipped to test P7 for Generator transient! Must add P7
to input streams');
end
%Pause action of local controller while %potential collision is resolved

self.actionpause
    = self.W + 1;

self.wait
    = self.W;

%Remember to check for collision %reversal transient

self.
collisioncheck
    = 1;

end

end

%Indicate that a transient is being analyzed

self.identifying = 1;

%Identify the transient for i = 1:length(self.
stream)

% %

transient.( self.

stream{i}) = self.series.(self.stream{i})(1:self.
analysis_loc + 2/self.tstep);

transient.( self.

stream{i}) = self.series.(self.stream{i})(1:end);

end

% [self.inds, adv_to] =

self.trans_search(self.series, []);

% if self.inds.P7 > 0 &&

self.inds.f > 0

self.ind = min([

self.inds.f, self.inds.P7]);

adv_to = min([

adv_to, self.inds.P7+floor(1/self.tstep)]);

% elseif self.inds.f > 0

self.ind = self.

inds.f;

.self.inds.P7 = self.

 inds.f;

else if self.inds.P7 > 0

self.ind = self.

inds.P7;

self.inds.f = self.

adv_to = self.inds.
P7+floor(1/self.tstep);

end
samples you advance

further analysis if the

anything this

happen if trans_search

just too far out in

time, and we

it. That in mind,

within a second or

location.

adv_to = min([adv_to, self.analysis_loc + floor(2/self.tstep)]);

[ID, ~, true_nd] = self.trans_class(self.series, self.ind)

just create a

the transient of

identified
if ID == 1
  self.num_ecu = [
    self.num_ecu(1)+1, self.num_ecu(1)];
  self.wait = true;

elseif ID == 2
  self.num_ecu = [
    self.num_ecu(1)-1, self.num_ecu(1)];
  self.wait = true;

elseif ID == 3
  self.num_gen = [
    self.num_gen(1)+1, self.num_gen(1)];
  self.wait = true;

elseif ID == 4
  self.num_gen = [
    self.num_gen(1)-1, self.num_gen(1)];
  self.wait = true;

elseif ID == 5
  self.wait = adv_to;
end

% Reset own-transient indicator
self.lookfor = 0;

% Reset transient index
indicator

% Reset "identifying"

self.ind = 0;

self.identifying = 0;

self.ind = 0;
self.identifying = 0;

else

% Move the transient to the desired position
self.wait = self.ind - self.analysis_loc;
self.actionpause = self.wait + 1;

for x = 1:length(self.stream)
    if self.inds.(self.stream{x}) > 0
        self.inds.(self.stream{x}) =
        max([1, self.analysis_loc + self.inds.
            self.stream{x}]) - self.ind
    end
end

end
self.ind = self.
analysis_loc;
end
else

%Set window to advance by the
preset amount
self.wait = self.adv_search;
end
end
else
ID = 0;
self.wait = self.wait - 1;
end
end
elseif self.init_wait > 0
self.init_wait = self.init_wait - 1;
selidentifying = 1;
sel.actionpause = 1;
else
self.init_wait = -1;
sel.actionpause = 0;
sel.freq_base = sel.find_base(self.series.f);
sel.identifying = 0;
end
end
function [inds, ind_nd] = trans_search(self, sig, indx)

for i = 1:length(self.stream)
if strcmpi(self.stream{i}, 'f')

%Set transient detection parameters
thresh = 25;
clust = 30;

%Verify that recorded dataset is actually a transient
sig_search = sig.(self.stream{i}) - self.
find_base(sig.(self.stream{i}));
[ind_st, ind_nd] = self.is_trans_f(
sig_search, 0.2, 5, 'narrow');
if ind_st == 0 && ind_nd == 0
    [ind_st, ind_nd] = self.is_trans_f(
        sig_search, 0.01, 240, 'wide');
end
inds.(self.stream{i}) = ind_st;

elseif strcmpi(self.stream{i}, 'P7')
%Find edge index, if there are any.
[inds.(self.stream{i}), ~, ~] = self.
is_trans_P7(sig.(self.stream{i}), indx);
end
end
end

function init(self, method, ctrl_mode, tstep, sig_delay, ID_delay)

%Determine whether or not logging exemplars
if strcmpi(ctrl_mode, 'generate exemplars') ||
  strcmpi(ctrl_mode, 'manual')
  self.exemplar_clear(self.load_increment);
  self.logging_exemplars = 1;
else
  self.logging_exemplars = 0;
end

self.W = ceil(self.Wlen/tstep);
  self.wait = self.W;
self.medsize.P7 = floor(1/tstep);
self.medsize.f = 1;
self.tstep = tstep;
self.init_wait = 3*self.W;
self.wait = 0;
for i = 1:length(self.stream)
  self.series.(self.stream{i}) = zeros(self.W,1);
end
self.exemplars = {};

if strcmpi(method, 'god')
  self.method = 'god';
  self.num_ecu = zeros(1, 2 + floor(0.66*self.Wlen/tstep));
  self.num_gen = self.Gen.N*ones(1, 2 + floor(0.66*self.Wlen/tstep));
  self.sensed = zeros(1, 2 + ceil(sig_delay/tstep));
elseif strcmpi(method, 'mortal')
self.method = 'mortal';
self.num_ecu = [0 0];
self.num_gen = [1 1];

elseif strcmpi(method, 'test')
    self.method = 'test';
    self.num_ecu = 0;
    self.num_gen = 1;
end

a = load('./Exemplar_Library/current_library');
self.exemplars = a.library;
clear a;
end

% Mid-level tools

function pos = is_trans(self, transient, stream)
    if strcmpi(stream, 'f')
        [pos, ~] = self.is_trans_f(transient, 0.2, 5, 'narrow');
        if ind_st == 0
            [pos, ~] = self.is_trans_f(transient, 0.01, 240, 'wide');
        end
    elseif strcmpi(stream, 'P7')
        [pos, ~, ~] = is_trans_P7(transient.(stream))
    end
end

function [ind_st, ind_nd] = is_trans_f(self, sig,
min_deflec, clust, search_mode)

ind_st = 0; % Initial value of transient index, indicates NO transient

ind_nd = ind_st;

if mean(sig) > 40
    0;
end

if ~(length(sig) == 1 && sum(sig) == 0)
    % Evaluate merit as transient (frequency change-of-mean method)
    % Gather signal information
    if isempty(self.freq_base)
        sig_base = self.find_base(sig);
    else
        sig_base = self.freq_base;
    end

    %
    %
    figure(1); clf;
    plot(sig - sig_base, 'k'); hold on; plot(1:length(sig), sig_std*thresh*ones(size(sig)), 'r');
    %
    pause(0);

    % Compute which samples are outside of the nondeviation bandwidth, erode
    % and dilate to eliminate discontinuous deviations
    sig = sig - sig_base;

    if mean(sig) < -30
deflec = abs(sig) > min_deflec;
deflec_denoised = self.dilate(self.erode(deflec, floor((clust-1)/2)), floor((clust-1)/2) + 0.5/self.tstep);

%Take the first 1 in the deflection vector as
%the beginning of the transient
if sum(deflec_denoised)
    bounds = diff(deflec_denoised);
st_bound = find(bounds==1,1); ind_st = st_bound;
bounds(1:st_bound) = 0;
nd_bound = find(bounds==-1,1); ind_nd = nd_bound;
deflec_denoised(1:st_bound) = 0;
deflec_denoised(nd_bound:end) = 0;

st_found = 0;
nd_found = 0;
i = 1;
while ~(st_found & nd_found) && i < floor(5/self.tstep);
    deflec_denoised = self.dilate(deflec_denoised, 1);
    if abs(sig(find(deflec_denoised,1))) < 0.05 & ~st_found
st_found = 1;
ind_st = find(deflec_denoised,1);
end
if abs(sig(find(deflec_denoised,1,'last'))) < 0.05 && ~nd_found
    nd_found = 1;
    ind_nd = find(deflec_denoised,1,'last');
end
i = i + 1;
end
ind_st = max([1, ind_st]);
ind_nd = min([length(sig), ind_nd]);

% Generate normalizing score for a .1Hz, 2 s frequency deflection. Ensure there's enough energy in the signal to be considered a distinct signal!
if strcmpi(search_mode, 'wide')
    norm = sqrt(sum(.01*ones(floor(4/self.tstep),1))) % CHANGED IF STRCMP(MODE, 'WIDE') TO 1 05/18/2018
end
% Score transient by normalizing its norm
% if ~(sqrt(sum(sig(ind_st:ind_nd).^2)) > norm) || 1
if strcmpi(search_mode, 'wide')
    plot(sig);
    title(['num2str(sqrt(sum(sig
    (ind_st:ind_nd).^2)), ' vs. ', num2str(norm))
    0;
    end
    ind_st = 0;
    ind_nd = 0;
end
end
end

function [ind, dir, strength] = is_trans_P7(self, sig , indx)

    %Set initial values (indicate no edge if unchanged)
    ind = 0;
    dir = 0;
    strength = 0;

    shape = [-ones(floor(.25/self.tstep),1); zeros
    (floor(.15/self.tstep),1); ones(floor(.25/self.tstep),1)];
    lag = floor(length(shape)/2);
    nldiff = conv(sig, shape, 'valid') / (floor(.4/
    self.tstep));
    check_ind = floor(linspace(1,length(nldiff)
    -1,10));
    i = 1;
found_edge = 0;

while i < length(check_ind) && found_edge == 0

  [val_candidate, ind_candidate] = max(abs(nldiff(check_ind(i):check_ind(i+1))));

  ind_candidate = ind_candidate + check_ind(i);

  if val_candidate >= 0.1

    st = max([1, ind_candidate - floor(1.5/self.tstep)]);

    nd = min([length(sig), ind_candidate + floor(1.5/self.tstep)]);

    if abs(mean(sig(ind_candidate:nd)) - mean(sig(st:ind_candidate))) > 0.1

      found_edge = 1;

      ind = ind_candidate + lag;

      try

        dir = sign(nldiff(ind_candidate));

      catch

        0;

      end

      strength = abs((val_candidate - 0.1) / 0.1);

    end

  end

end

i = i + 1;

end
%Find edge using maximum value of gauss
difference

[gdiff, lag] = self.gaussdiff(sig);

%Factor of .75 prevents second diff of transient from
%influencing the threshold

thresh = 10*std(gdiff(lag:self.analysis_loc-lag
- floor(1/self.tstep)));%std(gdiff(lag:self.analysis_loc-
lag));

thresh = 3*std(gdiff);%std(gdiff(lag:self.
analysis_loc-lag));

check_inds = floor(linspace(1,length(gdiff),10));
i = 1;
found_edge = 0;

while i < length(check_inds) && found_edge == 0
    [val_candidate, ind_candidate] = max(abs(gdiff(check_inds(i):check_inds(i+1))));
    ind_candidate = ind_candidate + check_inds(i);

    if val_candidate >= thresh
        st = max([1, ind_candidate - floor(1.5/self.tstep)]);
        nd = min([length(sig), ind_candidate + floor(1.5/self.tstep)]);
        if abs(mean(sig(ind_candidate:nd)) - mean(sig(st:ind_candidate))) > .1
            found_edge = 1;

else

end

if found_edge
    % other code
else
    % other code
end
ind = ind_candidate;
dir = sign(gdiff(ind_candidate));
strength = abs((val_candidate - thresh) / thresh);

end
end

i = i + 1;
end
end

function [ID, true_st, true_nd] = trans_class(self, transient, inds)

%Transient prep

% for i = 1:length(self.stream)
%    indX = indX + self.stream{i}
% end

disp(['Before Prep: i_f = ', num2str(inds.f), ', i_7 = ', num2str(inds.P7)]);

[inds_c, transient] = self.trans_prep(transient, inds);

for i = 1:length(self.stream)
    if inds.(self.stream{i}) > 0
        indX = indX + self.stream{i};
    end
end

disp(['Before Prep: i_f = ', num2str(inds.f), ', i_7 = ', num2str(inds.P7)]);

end

%Exemplar Selection

xmplrs = self.get_exemplars(self.load_rel, self.

375
if inds.f < 0
 0;
end

score = 10*ones(length(self.stream), length(self.class));
for i = 1:length(self.stream)
  score(i,:) = 10*match_threshold(self.stream{i});
end

match_thresh = score/10 * (1 + 0.5*(num_gen(1)-1));

sts = zeros(length(self.stream), length(self.class));
nds = sts;

for i = 1:size(score,2)
  has_exemplar = false;
  for q = 1:size(score,1)
    has_exemplar = has_exemplar || ~isempty(xmplrs{1,i}.(self.stream{q}));
  end
  if has_exemplar
    %For every stream in current event class
for q = 1:size(score,1)

% If current event class/stream IS supposed to have a significant transient
if ~isempty(xmplrs{1,i}.(self.stream{q}))
    sub_score = -ones(length(xmplrs{1,i}.(self.stream{q})), 1);
    sub_st = zeros(length(xmplrs{1,i}.(self.stream{q})), 1);
    sub_nd = sub_st;
    for s = 1:length(sub_score)
        if mean(transient.(self.stream{q}) < -10)
            0;
        end
        [sub_score(s), ~, sub_st(s), sub_nd(s)] = self.get_score(transient.(self.stream{q}), xmplrs{1,i}.(self.stream{q}{s}), self.stream{q}, inds, i, match_thresh(q,i));
        if i == 3
            disp(['freq. score is:
', num2str(sub_score(s))]);
        end
    end
end
end
sub_st = sub_st(find(
sub_score >= 0)); self.
analysis_loc];

sub_nd = [sub_nd(find(
    sub_score >= 0)); self.
analysis_loc];

sub_score = sub_score(find(
    sub_score >= 0));

core = [score(q, i), w = min([sub_score
    ; 10*match_thresh(q, i)]);

sts(q, i) = sub_st(w) -
    inds_c.(self.stream{q});

nds(q, i) = sub_nd(w) -
    inds_c.(self.stream{q});

%If current event class/stream is NOT
supposed to have a
%significant transient
else

    [score(q, i), ~, sts(q, i), nds(q
    , i)] = self.get_score(
        transient.(self.stream{q}),
        [], self.stream{q}, inds, i,
        match_thresh(q,i));

end
end

%If there is NO exemplar for this class at
current state,
%rule out that class. (e.g. gen off w/ only 1
gen. active)
else
score(q, i) = 10*match_thresh(q, i);
end
end
%

We rely on a weak signal to detect generators on in
%frequency... It's a useful indicator, but the
correlation score is never
%very good because of the SNR. Therefore, if the
P7 score(2,3) is
%good, we allow it to improve the frequency score
(1,3) a bit.
%This should be a machine learning algorithm's
job.
%

disp(['freq. score = ', num2str(score(1,3))]);
disp(['P7 score = ', num2str(score(2,3))]);
%
score(1,3) = 0.5*score(1,3)+0.5*score(2,3);
disp(['new freq. score = ', num2str(score(1,3))]);
%

Find event classes that match transients across
ALL streams

match_ind = find(sum(score <= match_thresh, 1)
== size(score,1));
% match_ind = find(mean(score,1) == self.
match_threshold);

match_candidates = score(:, match_ind);
st_candidates = sts(:, match_ind);
nd_candidates = nds(:, match_ind);
%If there are events whose exemplar streams ALL match the transient,
%pick the best match from among them
if ~isempty(match_candidates)
    [~, ID_ind] = min(sum(match_candidates, 1));
    ID = match_ind(ID_ind);
    true_st = st_candidates(ID_ind);
    true_nd = nd_candidates(ID_ind);
%If NO good match of exemplar to transient is found, mark as spurious
else
    ID = 5;

    true_st = self.analysis_loc;
    true_nd = self.analysis_loc;
end

figure(1);
plot(self.series.f);
names = self.class;
names(length(names)+1) = {'spurious'};
names(length(names)+1) = {'nascent'};
title(['IDed as ', names{ID}]);
disp(names{ID});
if ID == 5
    disp(score);
    pause(0);
elseif (ID == 1 || ID == 2 || ID == 3 || ID == 4)
    disp(score);
pause(0);
end
end

%Low-level tools

function eroded = erode(self, sig, deg)
    %Shift dimensions if necessary
    flipped = 0;
siz = size(sig);
    if siz(1) < siz(2)
sig = shiftdim(sig, 1);
    flipped = 1;
end

if deg > 0 %deg=0 means do nothing
    sig_rshift = circshift(sig, 1); sig_rshift(1) = 0;
sig_lshift = circshift(sig, -1); sig_lshift(end) = 0;
eroded = sig & sig_rshift & sig_lshift;
end

if deg > 1 %deg=1 means no further recursion necessary
    eroded = self.erode(eroded, deg-1);
end

if flipped == 1 %flip back to original dimensions
    if necessary
    eroded = shiftdim(eroded, 1);
end
% Shift dimensions if necessary
flipped = 0;
siz = size(sig);
if siz(1) < siz(2)
sig = shiftdim(sig, 1);
flipped = 1;
end

if deg > 0 % deg=0 means do nothing
    sig_rshift = circshift(sig, 1); sig_rshift(1) = 0;
sig_lshift = circshift(sig, -1); sig_lshift(end) = 0;
    dilated = sig | sig_rshift | sig_lshift;
end

if deg > 1 % deg=1 means no further recursion necessary
dilated = self.dilate(dilated, deg-1);
end

if flipped == 1 % flip back to original dimensions if necessary
dilated = shiftdim(dilated, 1);
end
end

function dilated = dilate(self, sig, deg)

function base = find_base(~, data)
num_bins = 501;
members = zeros(num_bins-1,1);
bins = linspace(min(data), max(data), num_bins);

for i = 1:num_bins-1
    members(i) = sum(data >= bins(i) & data < bins(i+1));
end

base_ind = find(abs(members) == max(abs(members)), 1);
base = 0.5*(bins(base_ind)+bins(base_ind+1));
end

function exemplar_slice = get_exemplars(self, load_rel, num_gen)
    % If exemplar library has not yet been loaded, load it
    if isempty(self.exemplars)
        load( './Exemplar_Library/current_library');
        self.exemplars = library;
    end
    % Compute which load index to pull exemplars from (limit to %1:length(self.exemplars)
    load_ind = min([size(self.exemplars,1), max([1, find(load_rel<=floor(100/size(self.exemplars,1))):floor(100/size(self.exemplars,1)):100,1])])
    end
if ~(num_gen == 1 || num_gen == 2)
    disp('You messed up! Wrong generator count .');
    num_gen = max([1, min([2, num_gen])]);
end

%Pull exemplars
exemplar_slice = self.exemplars(load_ind, :, num_gen);

%exemplar_slice{3}.f = {xmg};

%Add P7 info step_calc(self, trans_type, load_init, gen_init)
exemplar_slice{1,1}.P7 = cell(1);
if self.num_ecu(1) < self.tot_ecu
    exemplar_slice{1,1}.P7 = {self.step_calc('ecu on', load_rel, num_gen)};
else
    exemplar_slice{1,1}.P7 = {};
end

exemplar_slice{1,2}.P7 = cell(1);
if self.num_ecu(1) > 0
    exemplar_slice{1,2}.P7 = {self.step_calc('ecu off', load_rel, num_gen)};
else
    exemplar_slice{1,2}.P7 = {};
end
exemplar_slice{1,3}.P7 = cell(1);
if self.num_gen(1) == 1
    exemplar_slice{1,3}.P7 = {self.step_calc('gen
    on', load_rel, num_gen)};
else
    exemplar_slice{1,3}.P7 = {};
end

exemplar_slice{1,4}.P7 = cell(1);
if self.num_gen(1) == 2
    exemplar_slice{1,4}.P7 = {self.step_calc('gen
    off', load_rel, num_gen)};
else
    exemplar_slice{1,4}.P7 = {};
end
end

function exemplar_clear(self, load_increment)

%Save previous exemplar library, just in case
load('./Exemplar-Library/current_library', 'library');
save('./Exemplar-Library/saved_library', 'library');

%Create and save new, blank library
library = cell(100/load_increment, length(self.class), 3);
for i = 1:size(library, 1)
    for s = 1:length(self.class)
        % code
    end
end

for i = 1:size(library, 2)
    % code
end
end
for p = 1:size(library, 3)
    library{i,s,p} = struct();
    for q = 1:length(self.stream)
        library{i,s,p}.(self.stream{q}) = {
        library{i,s,p}.([self.stream{q},'
        _tracker']) = {};
    end
end
end
end

save('./Exemplar_Library/current_library', '
    library');
self.exemplars = library;
end

function exemplar_update(self,
    load_rel, class, num_gen, transient)
    if load_rel <= 100 && load_rel >= 0
        %Generate transient info for match scoring
        [self.inds] = self.trans_search(transient,
            []);
        trans_stor = transient;
        [inds_c, transient] = self.trans_prep(
            transient, self.inds);
        if self.inds.f < 0
            0;
        end
        for i = 1:length(self.stream)
            ...
if self.inds.(self.stream{i}) > 0
    self.inds.(self.stream{i}) = max([
        self.inds.(self.stream{i}) +
        inds_c.(self.stream{i}), 1]);
end
end

if class == 3
    transient.f = transient.f - mean(
        transient.f);
    transient.f = medfilt1(transient.f, 120,
        'truncate');
end

if self.inds.f < 0
    0;
end

%Retrieve relevant exemplars
exm = self.get_exemplars(load_rel, num_gen(1));
xmplrs = exm{1,class};
for i = 1:length(self.stream)
    if self.inds.(self.stream{i}) > 0
        if isempty(xmplrs.(self.stream{i}))
            %Add transient as new exemplar
            if strcmpi(self.stream{i}, 'p7')
                trans_plot = transient.P7;
                transient.(self.stream{i}) =
                    self.edge_dir(transient.(self.stream{i}), self.inds .P7);
            end
        end
    end
xmplrs.(self.stream{1}){1} =
  transient.(self.stream{1});

transient.(xmplrs.([self.stream{1}, '_tracker']){1}{1}) = {transient.(self.
  stream{1})};

else

  score = 5*match_threshold.(self.stream{1})(class)*ones(,
    length(xmplrs.(self.stream{1})
  ), 1);

  lag = score;

for s = 1:length(xmplrs.(self.
  stream{1})){score(s), lag(s), ~, ~] =
  get_score(transient.(self.
    stream{1}), xmplrs.(self.
      stream{1}){s}, self.
    stream{1}, self.ind,
      class, self.
    match_threshold.(self.
      stream{1})(class));

end

[match_val, match_ind] = min(
  score);

if strcmpi(self.stream{1}, 'p7')
  trans_plot = transient.P7;
  transient.(self.stream{1}) =
    edge_dir(transient.(self.
      stream{1}), self.ind,

if class == 2 && floor(load_rel) == 36

clf;
plot(trans_plot);
title(['ind. = ', num2str(self.inds.P7), ', step = ', num2str(transient.(self.stream{i}))]);

end

end

if match_val <= self.integration_tolerance.(self.stream{i})
  %If YES, average it in somehow
  lag = lag(match_ind);
  if lag > 0
    transient.(self.stream{i}) = [zeros(lag,1);
    transient.(self.stream{i})(1:end-lag)];
  elseif lag < 0
    transient.(self.stream{i}) = [transient.(self.stream{i})(abs(lag)+1:end);
    zeros(abs(lag),1)];
  end

  %Morph best-matching exemplar
with transient
xmplrs.(self.stream{i}){
    match_ind = 0.8*xmplrs.(self.stream{i}){match_ind} + 0.2*transient.(self.stream{i});
    xmplrs.|self.stream{i},'_tracker'|} {match_ind} {
        length(xmplrs.|self.stream{i},'_tracker'|) {match_ind} +1} = transient.(self.stream{i});
}
xmplrs.|self.stream{i},'_tracker'|} {match_ind} {
        length(xmplrs.|self.stream{i})} {1} = transient.(self.stream{i});
}

else

    %Add transient as new exemplar
    xmplrs.(self.stream{i}){
        length(xmplrs.(self.stream{i})) +1} = transient.(self.stream{i});
        xmplrs.|self.stream{i},'_tracker'|} {length(xmplrs.(self.stream{i}))} {1} = transient.(self.stream{i});
    }

end

end

load_ind = find(load_rel <= floor(100/size(}
self.exemplars{load_ind, class, num_gen} = xmplrs;
end

%Save previous exemplar library, just in case
try
    %
load('./Exemplar_Library/current_library', 'library');
    %
save('./Exemplar_Library/saved_library', 'library');
    %
catch
end

%library = self.exemplars;
%save('./Exemplar_Library/current_library.mat', 'library');
end

function exemplar_randomize(self)
    justones = 0;
    for i = 1:size(self.exemplars,1)
        for s = 1:size(self.exemplars,2)
            for q = 1:size(self.exemplars,3)
                for p = 1:length(self.stream)
                    for n = 1:length(self.exemplars{i, s, q}.(self.stream{p}))
                        %Compute # of transients at given load level/direction
                        num_trans = length(self.

391
% Compute number of transients taken for test exemplar
num_taken = ceil(num_trans/2);

% Randomly select from transients
taken_ind = randi([1, num_taken], 1, num_taken);

trans = 0;

% For every randomly-selected constituent of exemplar (p)
for x = 1:num_taken
    trans = trans + self.
    exemplars{i,s,q}.([ self.
        stream{p},'_tracker' ]){n}{
        taken_ind(x)}/
    num_taken;
end

if num_trans == 1
    justones = justones + 1;
end

% Store exemplar formed from randomly selected
%transients at the appropriate load level/direction
self.exemplars{i,s,q}(self.stream{p}){n} = trans;

end
end
end
end

end

%disp([num2str(justones), ' just-ones! ']);
end

function [inds_c, transient] = trans_prep(self, data, inds)
    for i = 1:length(self.stream)
        if strcmpi(self.stream{i}, 'f')
            %Remove signal base
            sig_base = self.find_base(data.f);
            if isempty(self.freq_base)
                sig_base = self.find_base(data.f);
            else
                sig_base = self.freq_base;
            end
            transient.(self.stream{i}) = data.(self.stream{i}) - sig_base;%self.find_base(data.(self.stream{i}));
        end

        %Low-pass filter the signal
        [transient.(self.stream{i}), ind_corr] = self.LPF(transient.(self.stream{i}),

393
\[
\pi/8, \text{self.medsize.f)};
\]

if \( \text{inds.f} > 0 \)

\[ \text{inds_c.f} = -\text{ind_corr}; \]

else

\[ \text{inds_c.f} = 0; \]

end

if mean(\text{transient.f}) < -20

0;

end

elseif strcmpi(\text{self.stream{i}}, 'P7')

\[ \text{before} = \text{floor}(1.5/\text{self.tstep}); \]

\[ \text{after} = \text{ceil}(1.5/\text{self.tstep}); \]

\%Cut out the section of data
\%around the transient

if \( \text{inds.P7} > 1 \&\& \text{inds.P7} < \text{length(data.(self.stream{i}))} \)

if \( \text{inds.P7} \leq \text{before} \)

\[ \text{before} = \text{ceil}(\text{inds.P7}/4); \]

end

if \( \text{inds.P7} \geq \text{length(data.(self.stream{i}))} - \text{after} \)

\[ \text{after} = \text{ceil}((\text{length(data.(self.stream{i}))}) - \text{inds.(self.stream{i})})/4); \]

end

data.(self.stream{i}) = data.(self.stream{i})(inds.(self.stream{i}) - \text{before} : \text{inds.(self.stream{i})} + \text{after});
inds_c.P7 = before - inds.P7;

% Remove signal mean
transient.(self.stream{i}) = medfilt1
    (data.(self.stream{i}) - mean(data
     .(self.stream{i})), self.medsize.
     P7, 'truncate');

else

% No transient detected. Cut out a
% section of data
% near the middle.
cut_ind = floor(length(data.(self.
    stream{i}))/2);
data.(self.stream{i}) = data.(self.
    stream{i})(cut_ind-before:cut_ind+
    after);

inds_c.P7 = ceil((after-before)/2) -
    inds.P7;

% Remove signal mean
transient.(self.stream{i}) = medfilt1
    (data.(self.stream{i}) - mean(data
     .(self.stream{i})), floor(0.5/self.
     tstep), 'truncate');

end

end
end
end

function [data, ind_corr] = LPF(~, inp, wc, medsize)

% Measure length of input data
len_unfilt = length(inp);
%Median filter input data
dat = medfilt1(inp,medsize,'truncate');

%Create a LPF impulse response
lpf = wc*sinc(wc*(-floor(length(inp)/20):floor(length(inp)/20)));

%Convolve data with LPF impulse response
fdat = conv(dat, lpf, 'valid');

%Measure length of filtered data
len_filt = length(fdat);

%Calculate necessary index correction
ind_corr = floor((len_unfilt - len_filt)/2);

data = fdat;

end

function [step] = edge_dir(self, sig, ind)
    before = floor(1.5/self.tstep);
    after = ceil(1.5/self.tstep);

    %Ensure that we don't exceed matrix dimensions
    if ind <= before + 1
        before = ind-1;
    end

    if ind >= length(sig)-after
        after = length(sig) - ind;
    end
function \( \text{dist} \) = \text{gauss}(\sim, \text{stdd}, \text{len})

\[
\text{mn} = \text{floor}(\text{len}/2);
\]
\[
\text{x} = 1:\text{len};
\]
\[
\text{dist} = (1 / (\text{stdd} \times \text{sqrt}(2\pi))) \times \exp(-((\text{x} - \text{mn}).^2 / (2 * \text{stdd}^2)));
\]

end

function \([\text{score}, \text{lag}, \text{pos_st}, \text{pos_nd}]\) = \text{get_score}(
\text{self}, \text{trans}, \text{exemplar}, \text{strm}, \text{inds}, \text{xmplr_class}, \text{match_thresh})

% Frequency match score
if strcmpi(strm, 'f')

end
pos_st = 1;
pos_nd = self.analysis_loc;
if ~isempty(exemplar)

%If checking against a GEN_ON exemplar, then subtracting the base doesn’t work well because the entire transient can happen at an offset from the base frequency... subtract the mean instead!
if xmplr_class == 3
    trans = trans - mean(trans);
    trans = medfilt1(trans, floor(0.03/self.tstep), 'truncate');
end

%If you are checking against a short exemplar, you needn’t include the entire analysis window of the transient or of the exemplar... Just cut out the data about 1s after the transient WOULD have ended if it really did match the exemplar
[ind_st, ind_nd] = self.is_trans_f(exemplar, 0.2, 5, 'narrow');
if ind_st == 0
    ind_st = 1;

398
end
if ind_nd == 0
    ind_nd = length(trans);
end

ind_cut = floor((ind_nd - ind_st) + 4/
    self.tstep);

trans_corr = trans(1:min([ind_cut + inds
    .(strm), length(trans)]));
trans_corr = trans_corr - self.find_base(
    trans_corr);

xmplr_corr = exemplar(1:min([ind_st +
    ind_cut, length(exemplar)]));
    xmplr_corr = xmplr_corr - self.
    find_base(xmplr_corr);

%ONLY look for matching shapes in the
    vicinity of the
%analysis location! (Frequency transient
    should start
%within +/- 1s of the analysis location)
if ind_st <= inds.(strm)
    maxlag = inds.(strm) - ind_st + floor
        (2/self.tstep);
else
    maxlag = ind_st - inds.(strm) + floor
        (2/self.tstep);
end

399
[score, lag] = self.bixcorr(trans_corr, xmplr_corr, maxlag);

% Sometimes generator on transients can have an
% incorrect offset applied to them, because they take up
% almost the entire window. If checking for a generator
% on transient, use the best correlation lag to refine
% your transient window and try one more time.
if xmplr_class == 3
    start_ind = max([1, lag]);
    end_ind = min([length(xmplr_corr) + lag, length(trans_corr)]);
    trans_corr = trans_corr(start_ind:end_ind) - mean(trans_corr(start_ind:end_ind));
figure(2); clf;
plot(xmplr_corr); hold on; plot(trans_corr);
[score, lag] = self.bixcorr(trans_corr, xmplr_corr, maxlag);
% disp(num2str(score));
0;
end

% This is the true position of the transient IN THE
% PREPARED DATA STREAM
pos_st = lag + ind_st;
pos_nd = lag + ind_nd;

else

%If there is no detectable transient, 
score is perfect
if ~(inds.f > 0)
score = 0;
lag = 0;

pos_nd = self.analysis_loc +
floor(2/self.tstep);

%If expecting NO transient but a 
 transient is
%detected, score is the signal norm, 
divided by
%to the norm of a .1Hz, 1s deflection
else

%Find the bounds of the significant 
portion of the 
% transient
[st, nd] = self.is_trans_f(trans,
0.2, 5, 'narrow');
if st == 0 && nd == 0

%If the initial frequency 
 transient search 
% returns nothing, look for a low/
 wide transient
%instead of a high/narrow
[st, nd] = self.is_trans_f(trans,
0.01, 240, 'wide');
end

if st == 0 && nd == 0
    %If both frequency transient searches fail, use
    %self.ind (earliest indication of a transient
    %in the window) to set the test segment.
    st = self.ind;
    nd = self.ind + floor(2/self.tstep);
end

%Generate normalizing score for a .1 Hz, 2s
%frequency deflection (deflection equal in
%magnitude to the normalizing pulse will be
%considered too large).

norm = sqrt(sum(.01*ones(floor(2/self.tstep),1)));

%Score transient by normalizing it’s norm
if st == 0
    st = 1;
end
if nd == 0
    nd = floor(2/self.tstep);
end
score = match_thresh * sqrt(sum(trans(st:nd).^2))/norm;
lag = 0;
pos_st = st;
pos_nd = nd;
end
end

%P7 match score
elseif strcmpi(strm, 'P7')
    lag = 0;
pos_st = 1;
pos_nd = floor(length(trans)/2);
if ~isempty(exemplar)
    loc_min = max([inds.(strm) - floor(1.5/self.tstep), floor(0.1/self.tstep) + 1]);
    loc_max = min([inds.(strm) + floor(1.5/self.tstep), length(trans) - floor(0.1/self.tstep)]);

    score = 1e3 * ones(1, loc_max - loc_min + 1);
n1 = sum(trans.*trans);

    figure(2);
    hold on;
    plot(trans);

    for loc = loc_min:loc_max
        try
            xmplr = [zeros(loc - 1, 1); exemplar * ones(length(trans) - loc + 1, 1)];

3403
xmplr = xmplr - mean(xmplr);
catch
    0;
end

figure(1);
plot(xmplr); hold on; plot(trans);
prod = sum(trans.*xmplr);
n2  = sum(xmplr.*xmplr);
score(loc-loc_min+1) = (n1 + n2 - 2* prod) / max([n1, n2]);
plot(xmplr);

end

[score, l2] = min(score);
score = 2*score;
pos_st = l2; pos_nd = l2;

else
    if inds.P7 > 0
        score = abs(self.edge_dir(trans, inds .P7))*2.5*match_thresh;
    else
        score = 0;
    end
end
end

function [score, lag] = bixcorr(~, sig1, sig2, maxlag)
auto_corr1 = max(xcorr(sig1, sig1));
auto_corr2 = max(xcorr(sig2, sig2));

[corrs, lags] = xcorr(sig1, sig2, maxlag);
[score1, maxind] = max(corrs);

score1 = score1/auto_corr1 - 1;
score2 = max(corrs)/auto_corr2 - 1;

score = (auto_corr1 + auto_corr2 - 2*max(corrs))
       / auto_corr2;

lag = lags(maxind);

function score = calc_error(~, target, actual)
    score = abs((target-actual)/0.75);
end

function step_size = step_calc(self, trans_type, load_rel, gen_init)
    load_init = self.load_rel();
    load_init_abs = load_rel*gen_init*self.C/100;

    if strcmpi(trans_type, 'ecu on')
        load_fin = 100*(load_init_abs + self.H_PWR) /
                   (self.C*gen_init);
        gen_fin = gen_init;
    elseif strcmpi(trans_type, 'ecu off')
        load_fin = 100*(load_init_abs - self.H_PWR) /
self.C*gen_init);
gen_fin = gen_init;

elseif strcmpi(transtype, 'gen on')
    load_fin = 100*load_init_abs / (self.C*(
gen_init + 1));
    gen_fin = gen_init + 1;
else if strcmpi(transtype, 'gen off')
    load_fin = 100*load_init_abs / (self.C*(
gen_init - 1));
    gen_fin = gen_init - 1;
end

step_size = self.est_P7(load_fin, gen_fin) - self
            .est_P7(load_rel, gen_init);
end

function val = est_P7(~, load, gen)
    val = sum([.5396 - 0.0124*load ,...
              .0017*load .^2 ,...
              -2.2924e-5*load .^3 ,...
              9.8223e-8*load .^4 ,...
              0*(gen == 1) ,...
              -0.663*(gen == 2)]);
end

function estload_adj = estload_adj(self, ecuadj)
    if strcmpi(self.method, 'god')
        estload_adj = self.MC.load + self.H_PWR*ecuadj;
    else
        estload_adj = (self.num_ecu(1) + ecuadj)*(
                        self.H_PWR) + self.tot_ecu*self.V_PWR;
end
B.5.5 latrine_pump.m

classdef latrine_pump < handle

    %LC Summary of this class goes here
    % Detailed explanation goes here

    properties (Access = private)
        turning_on
        sig3

        on_timer = -1;
        off_timer = -1;
        start_time = -1;

        Rl_off     = 1e5;
        Rl_steady  = 11;
        Rl_turnon  = 5;

end

    properties (Dependent)
        on

end

    properties
        Rl = 1e5;
enabled = 0;
end

methods

%Constructor method
function latrine_pump = latrine_pump()
end

%Getter methods
function is_on = get.on(self)
    is_on = 1;
  else
    is_on = 0;
  end
end

%High-level functions
function operate(self, t)
  if self.enabled
    if ~self.turning_on
      if ~self.on
        if self.start_time == -1 || self.on_timer == -1 || self.off_timer ~= -1
          self.start_time = t;
          self.off_timer = -1;
          self.on_timer = randi([300,600],1);
        end
      end
    end
  end
end
end

if t - self.start_time > self.on_timer
    self.on_timer = -1;
    self.start_time = -1;
    self.sig3 = 0;
    self.turning_on = 1;
end
end
else

%MANAGE PUMP TURN-ON
if self.sig3 > 0
    if t - self.sig3 < 0.1
        self.Rl = self.Rl_turnon;
    else
        self.Rl = self.Rl_steady;
        self.turning_on = 0;
    end
else
    self.sig3 = t;
end
end

if self.on
    if self.start_time == -1 || self.
        off_timer == -1 || self.on_timer >= -1
        self.start_time = t;
        self.on_timer = -1;
        self.off_timer = randi([10,100],1);
end
B.5.6 lc.m

The lc (local controller) module has multiple ECU control logics programmed in it, including autonomous, centralized, and thermostatic. Which of these modes it operates in is determined at the beginning of a simulation, when it is initialized.

classdef lc < handle
    %LC Summary of this class goes here
    % Detailed explanation goes here

    properties (Access = private)

        on_since
            max_ontime_o = 300;

end
max_ontime  = 300;

on_timer = -1;
off_timer = -1;
start_time = -1;

num

temp_ind

%OVERLOAD correction timer
max_wait_OL = 3;
max_deficit_OL = 6;
min_wait_OL = 1;
min_deficit_OL = -6;

%QUICK REACT timer
max_wait_quick = 200;
max_deficit_quick = 10;
min_wait_quick = 20;
min_deficit_quick = -10;

%SLOW REACT timer
max_wait_slow = 60;
max_deficit_slow = 2;
min_wait_slow = 10;
min_deficit_slow = -2.5 - 5.5;

ctrl_mode
resolving_collision = 0;

wait_coldstart
end

properties
  collision_times = [];  
commandeered = 0;

vent_state = 1;
heat_state = 0;

setpoint
temp_hbw = 1.667;
max_hesitation

tempvec

MC
Tent
identifier
end

methods
  %Constructor method
  function LC = lc(identifier, num)
    LC.identifier = identifier;
    LC.num = num;
  end
% Indicate tent of assigned duplex from which to read temp.
LC.temp_ind = 3 - mod(num, 2) * 2;
if mod(num, 2)
    LC.temp_ind = 1;
else
    LC.temp_ind = 3;
end

function init(self, setpoint, ctrl_mode)
    self.setpoint = setpoint;
    self.ctrl_mode = ctrl_mode;
    self.tempvec = zeros(600/self.identifier.tstep, 1);
    self.wait_coldstart = floor(1*3600/self.identifier.tstep);
end

% High-level functions
function decide(self, t)
    % Count down coldstart timer (prevents the autonomous control
    % from entering "desperate" mode for 30 minutes after coldstart.
    % In desperate mode, ecus are willing to turn on
    % in violation of
    % the peak load constraint.
    if self.wait_coldstart
        self.wait_coldstart = self.wait_coldstart -
1;

```python
else if self.wait_coldstart < 0
    self.wait_coldstart = 0;
end

if self.identifier.collision && ~self.resolving_collision
    %Undo the state change that caused the collision
    self.start_time = t;
    if self.heat_state
        self.off_timer = (self.identifier.Wlen) * randi([30,40]) / 100;
    else
        self.on_timer = (self.identifier.Wlen) * randi([60,70]) / 100;
    end

%Log collision time
self.resolving_collision = 1;
self.collision_times = [self.collision_times, t];
end

%OFF FIRST control
if strcmpi(self.ctrlmode, 'off first')
    self.decide_off_first(t);
%ON FIRST control
elseif strcmpi(self.ctrlmode, 'on first')
    self.decide_on_first(t);
```

414
%LEGACY control

elseif strcmpi(self.ctrl_mode, 'legacy')
    self.decide_legacy();
elseif strcmpi(self.ctrl_mode, 'centralized')
    self.commandeered = 1;
    self.heat_state = self.MC.heat_cmd(self.num);
end

function decide_on_first(self, t)
    %Call identifier to process data
    self.identifier.ID();

    %Commandeered operation (orders from master controller)
    if self.MC.commandeering
        self.commandeered = 1;
        %
        self.vent_state = self.MC.vent_cmd(self.num);
        self.heat_state = self.MC.heat_cmd(self.num);
    end

    %Autonomous operation (orders from internal control logic)
    else
        self.commandeered = 0;
        if ~(self.identifier.identifying || self.identifier.actionpause)
            %Turn-off (OVERLOAD)
            if self.identifier.is_overload && self.heat_state

        end

        end
if self.start_time == -1 || self.
    on_timer ~= -1
    self.start_time = t;
    self.on_timer = -1;
    self.off_timer = self.
    wait_overload('off');
end

if t - self.start_time > self.
    off_timer
    self.heat_state = 0;
    self.identifier.lookfor = 2;
end

%Turn-off (overtemp)
elseif self.Tent.state(self.temp_ind) >
    self.setpoint + self.temp_hbw && self.
    heat_state
    if self.start_time == -1 || self.
        on_timer ~= -1
        self.start_time = t;
        self.on_timer = -1;
        self.off_timer = self.wait_slow('off');
    end
end

if t - self.start_time > self.
    off_timer
    self.heat_state = 0;
    self.identifier.lookfor = 2;
end
%Turn-on (undertemp)

```plaintext
elseif self.Tent.state(self.temp_ind) <
    self.setpoint - self.temp_hbw && ~self.heat_state
    if self.start_time == -1 || self.off_timer == -1
        self.start_time = t;
        self.off_timer = -1;
        self.on_timer = self.wait_slow('on');
    end

    if t - self.start_time > self.on_timer
        self.heat_state = 1;
        self.identifier.lookfor = 1;
    end

%Do nothing (temperature deadband)
else
    self.start_time = -1;
    self.on_timer = -1;
    self.off_timer = -1;
end
else
    if self.identifier.actionpause > 0
        self.identifier.actionpause = self.identifier.actionpause - 1;
    end
    if self.start_time > 0
        self.start_time = self.start_time +
```
function decide_off_first(self, t)
    %Commandeered operation (orders from master controller)
    if self.MC.commandeering
        self.commandeered = 1;
        self.vent_state = self.MC.vent_cmd(self.num);
        self.heat_state = self.MC.heat_cmd(self.num);
    %Autonomous operation (orders from internal control logic)
    else
        self.commandeered = 0;
    %Log temperature
        self.tempvec = circshift(self.tempvec, 1);
        self.tempvec(1) = self.Tent.state(self.temp_ind)-self.setpoint;
    %Call identifier to process frequency data
        self.identifier.ID();
        if (~(self.identifier.identifying || self.identifier.actionpause) || self.resolving_collision)
%Turn-on (capacity available)
if ~self.heat_state && (self.identifier.
    room_for_one && (self.Tent.state(self.
    temp_ind) < self.setpoint - self.
    temp_hbw) || self.identifier.collision
)

    if self.start_time == -1 || self.
        on_timer == -1 || self.off_timer
        == -1

        self.start_time = t;
        self.off_timer = -1;
    if ~self.identifier.collision
        self.max_on_time = self.
            max_on_time_0 - 10*mean(
                self.tempvec);
        self.on_timer = self.
            wait_slow('on');

    end
end

if t - self.start_time > self.
    on_timer
    self.heat_state = 1;
    if self.identifier.collision
        self.identifier.collision = 0;
        self.resolving_collision = 0;
        self.collision_times = [self.
            collision_times, t];
    else

419
self.identifier.lookfor = 1;
end
self.on_since = t;
end

%Turn-on (Desperate)... IF compartment falls more than
%2 deg. outside of its comfort band AND the coldstart
%timer has expired AND turning on will not overload the
%the generators, it allows itself to violate the
%peak constraint.
elseif ~self.heat_state && ((mean(self.tempvec) < self.temp_hbw - 1) && ~self.wait_coldstart && ~self.identifier.willoverload)
if self.start_time == -1 || self.on_timer == -1 || self.off_timer == -1
    self.start_time = t;
    self.off_timer = -1;
    if ~self.identifier.collision
        self.max_on_time = self.
        max_on_time_o = 10*mean(self.tempvec);
        self.on_timer = randi((1,100),1)/10;
    end
if t - self.start_time > self.on_timer
    self.heat_state = 1;
    if self.identifier.collision
        self.identifier.collision = 0;
        self.resolving_collision = 0;
        self.collision_times = [self.collision_times, t];
    else
        self.identifier.lookfor = 1;
    end
    self.on_since = t;
end

%Turn-off (ovetemp)
elseif self.heat_state && ((self.Tent.
    state(self.temp_ind) > self.setpoint +
    self.temp_hbw) || self.identifier.
    collision)
    if self.start_time == -1 || self.
        off_timer == -1 || self.on_timer
        ~= -1
        self.start_time = t;
        self.on_timer = -1;
        if ~self.identifier.collision
            self.off_timer = self.
            wait_slow('off');
end
end

if t - self.start_time > self.off_timer
    self.heat_state = 0;
    if self.identifier.collision
        self.identifier.collision = 0;
        self.resolving_collision = 0;
        self.collision_times = [self.collision_times, t];
    else
        self.identifier.lookfor = 2;
    end
end

%Turn-off (timeout)
elseif self.heat_state && (t - self.on_since > self.max_on_time)
    if self.start_time == -1 || self.on_timer == -1
        self.start_time = t;
        self.on_timer = -1;
        self.off_timer = self.wait_slow('off');
    end

if t - self.start_time > self.off_timer
self.heat_state = 0;
self.identifier.lookfor = 2;

end

%Do nothing (temperature deadband)
else
    self.start_time = -1;
    self.on_timer = -1;
    self.off_timer = -1;
end
else
    if self.identifier.actionpause > 0
        self.identifier.actionpause = self.identifier.actionpause - 1;
    end
    if self.start_time > 0
        self.start_time = self.start_time + self.identifier.tstep;
    end
end
end

function decide_legacy(self)
    if self.MC.commandeering
        self.commandeered = 1;
        self.vent_state = self.MC.vent_cmd(self.num);
        self.heat_state = self.MC.heat_cmd(self.num);
    %Autonomous operation (orders from internal

423
control logic)

else

%Turn On

if ~self.heat_state && self.Tent.state(self.temp_ind) < self.setpoint - self.temp_hbw
    self.heat_state = 1;
elseif self.heat_state && self.Tent.state(self.temp_ind) > self.setpoint + self.temp_hbw
    self.heat_state = 0;
end

end

end

%Mid-level functions

function wait OL = wait_overload(self , dir)

if strcmpi(dir, 'off')
    deficit = self.setpoint - self.Tent.state(self.temp_ind);
elseif strcmpi(dir, 'on')
    deficit = self.Tent.state(self.temp_ind) - self.setpoint;
end

m = (self.max_wait OL - self.min_wait OL)/(self.max_deficit OL - self.min_deficit OL);
wait OL = self.min_wait OL + (deficit - self.min_deficit OL)*m;
wait OL = max([wait OL, self.min_wait OL]) + max([0.5*randn(1), -self.min_wait OL]);

end

function wait_quick = wait_quick(self, dir)
    if strcmpi(dir, 'off')
        deficit = self.setpoint - self.Tent.state(self.temp_ind);
    elseif strcmpi(dir, 'on')
        deficit = self.Tent.state(self.temp_ind) - self.setpoint;
    end

    m = (self.max_wait_quick - self.min_wait_quick)/(self.max_deficit_quick - self.min_deficit_quick);
    wait_quick = self.min_wait_quick + (deficit - self.min_deficit_quick)*m;

    wait_quick = max([wait_quick, self.min_wait_quick]) + max([2*randn(1), -self.min_wait_quick]);
end

function wait_slow = wait_slow(self, dir)
    if strcmpi(dir, 'off')
        deficit = self.setpoint - self.Tent.state(self.temp_ind);
    elseif strcmpi(dir, 'on')
        deficit = self.Tent.state(self.temp_ind) - self.setpoint;
    end

    m = (self.max_wait_slow - self.min_wait_slow)/(
self.max_deficit_slow = self.min_deficit_slow;

wait_slow = self.min_wait_slow + (deficit - self.min_deficit_slow)*m;

wait_slow = wait_slow + 0.5*randn(1);

if wait_slow <= 0
    %MODIFIED TO MAKE COLLISIONS MUCH MORE LIKELY
    %change randi
    %first argument back to 1e2 to restore default behavior
    wait_slow = self.min_wait_slow * randi([1e2,2e3]) / (1e3);
end

disp(['temp is ', num2str(self.Tent.state(self.temp_ind)), ' wait is ', num2str(wait_slow), ' to turn ', dir]);
end

end

B.5.7 mc.m

The master controller serves to implement the centralized control algorithms developed in this thesis, generate exemplars for use by the identifier modules, and to monitor operation of the camp's generators and loads.

classdef mc < handle
    %MC Summary of this class goes here
    % Detailed explanation goes here

426
properties

ID = 'mc';

mode

commandeering = 0;

setpoint

Tent

Gen

ECU

decision_period = 12; %Decision period (minutes)
decision_len

ind = 0;
sched

heat_cmd

P = 0

I = 0.004

err
duty_integral = 0;
temp_av

max_simultaneous
duty_rec

adj
duty_max

%Variables for exemplar generation

generating_exemplars

exemplar_timer = 20;
exemplar_space
exemplar_count = 1;
ecu_count = 1;
onoff = 1;
end

properties (Dependent)
    num_ecu
    num_gen
    load
end

methods
    %Class constructor
    function MC = mc(Tent, Gen, ECU)
        MC.Tent = Tent;
        MC.Gen = Gen;
        MC.ECU = ECU;
        MC.heat_cmd = zeros(1, length(MC.ECU));
        MC.temp_av = zeros(1, length(MC.ECU));
    end

    %Getter functions
    function num_ecu = get.num_ecu(self)
        num_ecu = 0;
        for i = 1:length(self.ECU)
            num_ecu = num_ecu + self.ECU{i}.LC.heat_state
        end
    end
function num_gen = get.num_gen(self)
    num_gen = self.Gen.N_next;
end

function load = get.load(self)
end

function set.duty_integral(self, val)
    val(find(val > 100)) = 100;
    val(find(val < -100)) = -100;
    self.duty_integral = val;
end

%Simulation functions
function init(self, mode, tstep, setpoint, max_simultaneous)
    self.mode = mode;

    if strcmpi(mode, 'centralized')
        self.commandeering = 1;
        self.generating_exemplars = 0;
    elseif strcmpi(mode, 'generate exemplars')
        self.commandeering = 1;
        self.generating_exemplars = 1;
    elseif strcmpi(mode, 'autonomous')
        self.commandeering = 0;
        self.generating_exemplars = 0;
    elseif strcmpi(mode, 'manual')
        self.commandeering = 1;
        self.generating_exemplars = 0;
    end
self.decision_len = floor(60*self.decision_period/tstep);
self.setpoint = setpoint;
self.max_simultaneous = max_simultaneous;
self.I = self.I*tstep;
self.duty_integral = zeros(1,length(self.ECU));
self.duty_max = round(100*(self.max_simultaneous/length(self.ECU))) * ones(1,length(self.ECU));
self.exemplar_space = floor(self.exemplar_time/tstep);
self.duty_rec = zeros(1,length(self.ECU));
end

function set_schedule(self, duty_cycles, num_timesteps, current_state)
%Ensure all requested duty cycles are within [0%, 100%]
for i = 1:length(duty_cycles)
   if duty_cycles(i) < 0
      duty_cycles(i) = 0;
   elseif duty_cycles(i) > 100
      duty_cycles(i) = 100;
   end
end

%Clip duty cycles proportionally to satisfy max simultaneous
%run requirement

if sum(duty_cycles/100) > self.max_simultaneous
    duty_cycles = duty_cycles*(100*self.
    max_simultaneous/sum(duty_cycles));

    duty_cycles = duty_cycles - mod(duty_cycles
    ,100/self.decision_len);

    self.duty_integral = duty_cycles - self.P.*
    self.err;

end

%Impose maximum duty cycle saturation, and
adjustment of
%maximum duty cycle parameter to attain equal
temperature
%errors between tents

%Compute how maximum duty cycle parameters should
be adjusted
self.adj = (self.
    err - mean(self.err))*5;

%Adjust maximum duty cycles
self.duty_max = self.
    duty_max + self.adj;

%Reduce new maximum duty cycles as needed to
ensure
%compatibility with duty cycle discretization.
duty_cycles(duty_cycles > self.duty_max) = self.
    duty_max(duty_cycles > self.duty_max);
% If doing pure-integral feedback control, impose saturation on error integral
if self.P == 0
    self.duty_integral = duty_cycles;
end

% disp(num2str(duty_cycles));
self.duty_rec = duty_cycles;

% Convert requested duty cycles to requested # of timsteps ON for each ECU
steps_req = round(num_timesteps* duty_cycles/100);

if sum(steps_req) > (num_timesteps* self.max_simultaneous)
    t = 1;
    dng = sum(steps_req) - ceil(num_timesteps* self.max_simultaneous);
    for i = 1:dng
        steps_req(t) = steps_req(t) - 1;
        t = t + 1;
        if t > length(self.ECU)
            t = 1;
        end
    end
end

% Construct empty run-time schedule to be filled
self.sched = zeros(num_timesteps,
    length(steps_req));

%Instantiate a counter to indicate last assigned runtime slot
fin = 0;

%Minimum overlap method
for i = 1:length(steps_req)
    start = fin + 1;
    if start + steps_req(i) - 1 <= num_timesteps
        %Fill it out!
        self.sched(start:start + steps_req(i) - 1, i) = ones(steps_req(i), 1);
        fin = start + steps_req(i) - 1;
    else
        %Fill out to the end, then loop back to the beginning!
        self.sched(start:num_timesteps, i) = ones(num_timesteps - start + 1, 1);
        self.sched(1:steps_req(i) - num_timesteps + start - 1, i) = ones(steps_req(i) - num_timesteps + start - 1, 1);
        fin = steps_req(i) - num_timesteps + start - 1;
    end
end

rnk = zeros(length(self.sched(:, 1)), 1);

433
% As a first experiment, shift the schedule so that the first instance with
% the highest rank is brought to the front
for i = 1:length(self.sched(:,1))
    for s = 0:length(self.sched(1,:))
        if sum(self.sched(i,:) == current_state) == s
            % Store rank of schedule instance (how many device states match
            % last state of previous schedule)
            rnk(i) = s;
        end
    end
end

mx = find(rnk == max(rnk), 1);
sched = circshift(sched, [-mx+1, 0]);

% Another possibility is to minimize a cost function for the
% given schedule, where the scheduler is incentivized to run
% first the ecus whose tents most need heat
% for i = 1:size(sched, 1)
%    for s = 1:size(sched, 2)
%        rnk(i) = rnk(i) + self.err(i)*find(sched(:,s),1);
%    end
% end
end
function decide(self)
    if self.generating_exemplars
        self.generate_exemplars();
    elseif self.commandeering && strcmpi(self.mode, 'manual')
        if self.ind == 0
            %Initialize duty cycle requests
            duty = zeros(1, length(self.ECU));

            %Compute initial duty cycle requests
            for i = 1:length(duty)
                tent_ind = (i+mod(i,2))/2;
                temp_ind = 3-mod(i,2)*2;

                self.err(i) = self.setpoint - self.Tent{tent_ind}.state(temp_ind);

                duty(i) = min([self.P*self.err(i), 100]);
            end

            %Create a new schedule
            self.set_schedule(duty, self.decision_len, zeros(1,length(self.ECU)));

            %Set schedule index to 1
            self.ind = 1;
        else
            for i = 1:length(self.ECU)
if self.ind -- self.decisionlen

%Compute new duty cycle
self.err = self.setpoint - self.Tent{self.ind}.state(temp_ind);
self.duty_integral(i) = self.duty_integral(i) + self.I*self.err(i);

tent_ind = (i+mod(i,2))/2;
temp_ind = 3-mod(i,2)*2;
self.err(i) = self.setpoint - self.Tent{tent_ind}.state(temp_ind);
self.duty_integral(i) = self.duty_integral(i) + self.I*self.err(i);

%Update avg. temperature
self.temp_av(i) = self.temp_av(i) + self.Tent{tent_ind}.state(temp_ind)/self.decision_len;

%Execute runtime schedule
self.heat_cmd(i) = self.sched(self.ind, i);
end

if sum(self.sched(self.ind,:)) > self.max_simultaneous
    pause(0);
end

if self.ind == self.decision_len

%Compute new duty cycle
self.err = self.setpoint - self.temp_av;
self.temp_av = zeros(size(self.temp_av));
duty = self.P*self.err + self.duty_integral;

end
duty(find(duty > 100)) = 100;
duty(find(duty < 0)) = 0;
%Make a new schedule
self.set_schedule(duty, self.
decision_len, zeros(1,length(self.
ECU)));
plot(sum(self.sched,2));
%Reset index to 1
self.ind = 1;
else
self.ind = self.ind + 1;
end
end
end

function generate_exemplars(self)
if self.exemplar_count == self.exemplar_space
%Do a thing
if self.onoff == 1
%Turn on current tent
self.heat_cmd(self.ecu_count) = 1;
else
%Turn off current tent
self.heat_cmd(self.ecu_count) = 0;
end
if self.ecu_count == length(self.ECU)
self.ecu_count = 1;
self.onoff = -self.onoff;
else
self.ecu_count = self.ecu_count + 1;
end
end
B.5.8 weather

The weather module simply creates a time series of environmental temperature and solar irradiation for use in thermal simulation of the duplex modules.

```matlab
end
self.exemplar_count = 1;
else
  %Do a different thing
  self.exemplar_count = self.exemplar_count + 1;
end
function force_genon(self)
  self.Gen.force_on = {1, self.ID};
end
function force_genoff(self)
  self.Gen.force_off = {1, self.ID};
end
function force_genmanual(self)
  self.Gen.auto_dispatch = {false, self.ID};
end
end
```

```matlab
classdef weather < handle
  %WEATHER Summary of this class goes here
  % Detailed explanation goes here

  properties
    temp_series
```

438
rad_series

rad_bright = 50;

end

methods

function weather = weather()
end

function val = temp(self, t)
val = self.temp_series(find(abs(self.t - t) == min(abs(self.t-t)), 1));
end

function val = rad(self, t)
val = self.rad_series(find(abs(self.t - t) == min(abs(self.t-t)), 1));
end

function init(self, t, temp_mean, temp_var, sun)

% Initialize own time vector
self.t = t;

% Setup temperature vector
if strcmpi(temp_var, 'steady')
    self.temp_series = temp_mean*ones(size(self.t));
elseif strcmpi(temp_var, 'increase')
    self.temp_series = linspace(temp_mean-3, temp_mean+3, length(self.t));
elseif strcmpi(temp_var, 'decrease')
    self.temp_series = linspace(temp_mean+3,
temp_mean - 3, length(self.t));

elseif strcmpi(temp_var, 'bump')
    w = linspace(-pi/2, 3*pi/2, length(self.t));
    self.temp_series = temp_mean*ones(size(self.t)) + 2*(sin(w)-1);
elseif strcmpi(temp_var, 'dip')
    w = linspace(-3*pi/2, pi/2, length(self.t));
    self.temp_series = temp_mean*ones(size(self.t)) + 2*(sin(w)-1);
end

% Setup solar irradiance vector
if strcmpi(sun, 'bright')
    self.rad_series = self.rad_bright*ones(size(self.t));
elseif strcmpi(sun, 'night')
    self.rad_series = zeros(size(self.t));
elseif strcmpi(sun, 'increase')
    self.rad_series = linspace(0.75*rad_bright, 1.25*rad_bright, length(self.t));
elseif strcmpi(sun, 'decrease')
    self.rad_series = linspace(1.25*rad_bright, 0.75*rad_bright, length(self.t));
elseif strcmpi(sun, 'bump')
    w = linspace(-pi/2, 3*pi/2, length(self.t));
    self.rad_series = self.rad_bright*ones(size(self.t)) + 0.1*self.rad_bright*(sin(w) + 1);
elseif strcmpi(sun, 'dip')
    w = linspace(-3*pi/2, pi/2, length(self.t));
self.rad_series = temp_mean*ones(size(self.t)) + 0.1*self.rad_bright*(sin(w) - 1);

end

end

end

B.5.9 camp.m

The camp.m module ties together all of the modules for simulation purposes, ensuring that the information of each module is linked to the other modules that need it. It also conducts the simulation of all modules using the ode23 function and controls the time step at which each module is simulated.

classdef camp

    %A simulation testbed for HVAC control on small microgrids
    % The "camp" class serves as a structure for integrating various
    % models pertaining to HVAC systems in small microgrids.
    % These include: 1) Generation resources, 2) HVAC units, 3) HVAC controllers, 4) Tent thermal systems.

    properties
        ID = 'camp';
        last_min = 0;

end
%Set up camp elements
num_tent

%Connected objects
Weather
ECU
Gen
Duplex
MC
latrine_pump

%Choose HVAC control method
ctrl_mode
LC_mode

methods

function camp = camp(num_tent)
  %Set specified camp configuration
  if nargin > 0
    if mod(num_tent,2) ~= 0
      disp('Warning! Uneven number of tents specified. Increasing # by 1...');
      num_tent = num_tent + 1;
    end
    camp.num_tent = num_tent;
  else
    camp.num_tent = 4;
  end
%Set up weather system
Weather = weather(); camp.Weather = Weather;

%Set up transient identifiers for each ECU
Controller
Identifier = cell(num_tent, 1);
for i = 1:num_tent
    Identifier{i} = identifier(num_tent, i);
end

%Set up local controllers for each ECU
LC = cell(num_tent, 1);
for i = 1:num_tent
    LC{i} = lc(Identifier{i}, i);
end

%Set up ECUs (one for each tent compartment)
ECU = cell(num_tent, 1);
for i = 1:num_tent
    ECU{i} = ecu(LC{i});
end
%Connect ECUs to the camp
camp.ECU = ECU;

%Instantiate confederate load (latrine pump)
camp.latrine_pump = latrine_pump();

%Set up generator bank
Gen = gen(ECU, camp.latrine_pump);
%Connect generator bank to the camp
camp.Gen = Gen;

%Set up duplexes (adjoint tent couples)
Duplex = cell(num_tent/2, 1);
for i = 1:length(Duplex)
    Duplex{i} = duplex(ECU{2*i-1}, ECU{2*i}, Weather, i);
end

%Attach duplexes to camp
camp.Duplex = Duplex;

%Setup master controller
MC = mc(Duplex, Gen, ECU);
%Attach master controller to camp
camp.MC = MC;

%Make final intercamp connections
for i = 1:num_tent
    %Connect Identifiers to master controller and generator
    Identifier{i}.MC = MC; Identifier{i}.Gen = Gen;
    %Connect each local controller to master controller and to its respective duplex
    LC{i}.MC = MC; LC{i}.Tent = Duplex{floor(i/2) +mod(i,2)};
    %Connect each ECU to the generator
    ECU{i}.Gen = Gen;
end
end
end

function result = sim(camp, ctrl_mode, ID_mode, setpoint, temp_mean, temp_var, sun, t_fin, t_step, ID_delay, sig_delay, max_simultaneous, pump_enabled)

%%% Initialize Simulation

%Make time vector

\[ t = 0 : t_{\text{step}} : t_{\text{fin}} ; \]

%Make results cell

\[ \text{result} = \text{struct}('f', \text{zeros(length}(t),1), 'iv', \text{zeros(length}(t),3), 'load_est', \text{zeros(length}(t),\text{length}(\text{camp.ECU})), 'gen_est', \text{zeros(length}(t),\text{length}(\text{camp.ECU})), \ldots \]

\[ 'ecu\_state', \text{zeros(length}(t),\text{camp.num\_tent}), 'tent\_temp', \text{zeros(length}(t),\text{camp.num\_tent}), 'latrine\_pump\_state', \text{zeros(length}(t),1), \ldots \]

\[ 'env\_temp', \text{zeros(length}(t),1), 'num\_gen', \text{zeros(length}(t),1), 'duty', \text{zeros(length}(t),\text{length}(\text{camp.ECU})), \ldots \]

\[ 'S1', \text{zeros(length}(t),1)) ; \]

%%% Initialize each local controller's setpoint and identifier module

for i = 1:camp.num_tent

\[ \text{camp.ECU}\{i\}.LC.\text{identifier}\_\text{init}(ID\_mode, ctrl\_mode, t\_\text{step}, \text{sig\_delay}, \text{ID\_delay}) ; \]

\[ \text{camp.ECU}\{i\}.LC.\text{init}(\text{setpoint, ctrl\_mode}) \]
%Choose ECU control mode (centralized, autonomous)
camp.MC.init(ctrl_mode, t_step, setpoint, max_simultaneous);
camp.ctrl_mode = ctrl_mode;

%Initialize weather system
camp.Weather.init(t, temp_mean, temp_var, sun);

%Initialize tents
for i = 1:camp.num_tent/2
    camp.Duplex{i}.state = [camp.Weather.temp_series(1), 0, camp.Weather.temp_series(1), 0];
end

%Initialize latrine pump (enabled or disabled?)
camp.latrine_pump.enabled = pump_enabled;

%Iterate through time vector and simulate camp
simwait = 1/t_step - 1;
wait = 0;

for i = 2:length(t)
    min = (t(i)-mod(t(i),60))/60;
    disp(num2str(t(i)));
    if min ~= camp.last_min
disp(num2str(min));
camp.last_min = min;
end

%% Generators
%Simulate generators
[~,yout] = ode23(@(tt,yy)camp.Gen.gensim(tt, yy), [t(i-1) t(i)], [camp.Gen.state]);

%Update generator states
camp.Gen.state = yout(end,:);

%Store desired outputs
result.f(i) = yout(end,3); clear tout; clear yout;
result.S1(i) = camp.Gen.S1;
result.num_gen(i) = camp.Gen.N;

%% HVACs/Controllers
%Simulate HVACs/Controllers
camp.MC.decide();

%Simulate operation of latrine pump
confederate load
camp.latrine_pump.operate(t(i));
result.latrine_pump_state(i) = camp.latrine_pump.on;

for s = 1:camp.num_tent
    camp.ECU{s}.LC.decide(t(i));
result.ecu_state(i, s) = camp.ECU{s}.LC.heat_state;
result.load_est(i, s) = camp.ECU{s}.LC.identifier.num_ecu(1);
result.gen_est(i, s) = camp.ECU{s}.LC.identifier.num_gen(1);
end
result.duty(i, :) = camp.MC.duty_rec;

%% Tents
if wait == 0
for s = 1:camp.num_tent/2

%Set Duplex inputs
camp.Duplex{s}.temp = camp.Weather.temp(t(i));
camp.Duplex{s}.rad = camp.Weather.rad(t(i));

%Store desired outputs
if simwait <= length(result.tent_temp) - i - 1

%Store desired outputs
result.tent_temp(i:i+simwait,s*2-1) = camp.Duplex{s}.state(1)*ones(simwait+1,1);
result.tent_temp(i:i+simwait,s*2) = camp.Duplex{s}.state(3)*
%Simulate Duplexes
[~,yout] = ode23(@(tt,yy)camp.
    Duplex{s}.tentsim(tt,yy), [t(i - 1) t(i + simwait)], camp.Duplex
    {s}.state);

%Update duplex states
camp.Duplex{s}.state = yout(end
    ,:);

else
    try
        %Simulate Duplexes
        [~,yout] = ode23(@(tt,yy)camp
            .Duplex{s}.tentsim(tt,yy),
            [t(i - 1) t(end)], camp.
            Duplex{s}.state);

        %Update duplex states
camp.Duplex{s}.state = yout(end
            ,:);

        result.tent_temp(i + 1:end, s
            *2-1) = yout(end,1)*ones(
            length(result.tent_temp) -
            i, 1);
        result.tent_temp(i + 1:end,s*2)
            = yout(end,3)*ones(1,
length(result.tent_temp) -
i, 1);
catch
    pause(0)
    end
end
clear tout; clear yout;
wait = simwait;
end
else
    wait = wait - 1;
end
end
if strcmpi(ctrl_mode, 'generate exemplars') ||
    strcmpi(ctrl_mode, 'manual')
    try
        load('./Exemplar_Library/current_library', 'library');
        save('./Exemplar_Library/saved_library', 'library');
catch
    pause(0);
end
library = camp.ECU{1}.LC_identifier.exemplars;
save('./Exemplar_Library/current_library.mat', 'library');
end
B.5.10 test_sim.m

The test_sim.m script determines the parameters of each simulation, initializes the camp object, and logs the results of each simulation in the Results folder located in its directory. Along with the test results, it also saves a copy of every simulation module so that the effect of changing different module parameters and behaviors can be remembered.

```matlab
clear all;
clf; close all;
clear classes;

% num_iterations = {8,4,4,8};
files = {'camp.m', 'duplex.m', 'ecu.m', 'gen.m', 'identifier.m', 'lc.m', 'mc.m', 'test_sim.m', 'weather.m', 'test_sim.m'};

% loads = {linspace(1e3,4e4,40), linspace(1e3,4e4,40), linspace(2.5e3,9.8e4,80), linspace(1e3,5.5e4,40)};
ngen = {1};
num_iterations = {1};
pump_enabled = {1};
loads = {1e3};
end_t = {4*3600};

for i = 1:length(ngen)
    for s = 1:length(loads{i})
        for q = 1:num_iterations{i}
```
t_end = end_t[i];
tstep = .01;
ID_delay = 4;
sig_delay = 0;
ID_delaysize = ID_delay/tstep;
sig_delaysize = sig_delay/tstep;
setpoint = 21;
ctrl_mode = 'off first';
ID_mode = 'mortal';
environment_temp = {12, 'steady'};
solar_rad = 'night';
max_simultaneous = 4;

num_tent = 8;

Camp = camp(num_tent);

Camp.MC.force_genmanual();
Camp.Gen.base_load = {120^2*3/(loads{i}(s)), Camp.ID};
if ngen{i} == 0
    Camp.Gen.N = 1;
    Camp.Gen.N_next = 1;
    Camp.Gen.N_last = 1;
    Camp.MC.force_genon();
elseif ngen{i} == 3
    Camp.Gen.N = 2;
    Camp.Gen.N_next = 2;
    Camp.Gen.N_last = 2;
    Camp.MC.force_genoff();
else
Camp. Gen. N = ngen{i};
Camp. Gen. N_next = ngen{i};
Camp. Gen. N_last = ngen{i};

end

%Run simulation
result = Camp.sim(ctrl_mode, ID_mode, setpoint, environment_temp{1}, environment_temp{2},
                   solar_rad, t_end, tstep, ID_delay, sig_delay, max_simultaneous, pump_enabled{i});

%Save collision times
collision_times = cell(1, length(Camp.ECU));
for p = 1:length(Camp.ECU)
collision_times{p} = Camp.ECU{p}.LC.
collision_times;
end

if ~strcmpi(ctrl_mode, 'generate exemplars')

%Plot tent temperatures
figure(1);
plot(0:tstep:t_end, result.tent_temp*1.8 + 32);
xlabel('Time (s)'); ylabel('Temp (^oF)');
title('Tent Temperatures')
leg = cell(1, num_tent);
for p = 1:length(leg)
    leg{p} = ['Tent ', num2str(p)];
end
legend(leg);

%Plot load profile
figure(2);
plot(0:tstep:t_end, sum(result.ecu_state, 2));
xlabel('Time (s)'); ylabel('#ECUs Heating');
title('ECU Load Profile')

%Plot frequency
figure(3);
plot(0:tstep:t_end, result.f/(2*pi));
xlabel('Time (s)'); ylabel('Line Frequency (Hz)');
title('Line Frequency Simulation')

load testnum
folder_name = sprintf('./Results/Test %d', testnum);
testnum = testnum+1; save('./testnum', 'testnum');
mkdir(folder_name);
for i = 1:length(files)
    copyfile(files{i}, [folder_name, '/'], files{i});
end
save([folder_name, '/test_result'], 'result');
save([folder_name, '/collision_times'], 'collision_times');
end
B.6 MATLAB voltage data analysis/display

There are many scripts involved in analyzing and viewing voltage data gathered by the contact NILM.

B.6.1 main_analysis_tabular.m

This script iterates through enormous voltage measurement datasets collected by a NILM and generates a corresponding vector of frequency and harmonic amplitude.

```matlab
%% Sweep up, Set Dataset Parameters
clear;
close all;

Fs = 8000;

%% Specify Dataset Filename
filename = 'C:\Users\2016\Dropbox (MIT)\BCIL\Tests\AUG 31 2017\MIT Log\NILM\iv.txt';
filename = 'C:\Users\2016\Dropbox (MIT)\BCIL\Tests\SEP 12 2017\MIT Log\NILM Data\iv.txt';
filename = 'C:\Users\2016\Dropbox (MIT)\BCIL\Tests\SEP 14 2017\MIT Log\NILM Data\iv.txt';
filename = 'C:\Users\2016\Dropbox (MIT)\BCIL\Tests\SEP 26 2017\Contact Data\iv.txt';
filename = {'F:\31AUG17\iv.txt', 'F:\12SEP17\_iv1.txt', 'F:\12SEP17\_iv2.txt', 'F:\12SEP17\_iv3.txt'};
```
Wo Run Pete_prep on tabular data store

% Measure size of tabular data file
len = count_rows(test_td);
disp('done counting rows!');

% Conservatively (oversize) preallocate data arrays
S1 = -1e3*ones(round(2.5*len/Fs*60),8); S3 = S1;
f1 = -1e3*ones(round(2.5*len/Fs*60),1); f3 = f1;
tvect1 = -1e3*ones(round(2.5*len/Fs*60),1); tvect3 = tvect1;
Lvect1 = -1e3*ones(round(2.5*len/Fs*60),1); Lvect3 = Lvect1;
Fsvect1 = -1e3*ones(round(2.5*len/Fs*60),1); Fsvect3 = Fsvect1;

for q = 1:length(filename)
    test_td = tabularTextDatastore(filename{q});
    test_td.ReadSize = Fs*30; % Read 30s of data at a time
    test_td.ReadSize = Fs*30; % Read 30s of data at a time
end
ind = 1;
while hasdata(test_td)
    %Extract data from tabular file
    test = table2array(read(test_td));
    time = test(:,1);
    V1 = test(:,3);
    V3 = test(:,2);
    
    % disp(["reading from ", datestr(unix2matlab_EST(time(1))), ", to ", datestr(unix2matlab_EST(time(end)))]);
    run = 1;
    if unix2matlab_EST(time(1)) > 1.505225463012702e+15 || unix2matlab_EST(time(end)) < 1.505225463012702e+15
        run = 0;
    end

    if run == 1
        %Create data structure for passing test data to functions
        datal.Fs = Fs; data3.Fs = Fs;
        datal.time = time; data3.time = time;
        datal.Va = V1; data3.Va = V3;
        
        %Get zero-intercepts of Va and calculate average frequency
        datal = Zero_Intercept_Times(datal);
        data3 = Zero_Intercept_Times(data3);
        
        %Allocate to local variable for analysis
intercept_times1 = data1.intercept_times;
intercept_times3 = data3.intercept_times;

intercept_directions1 = data1.intercept_directions;
intercept_directions3 = data3.intercept_directions;

f_avg1 = mean(1./(2*diff(intercept_times1)));
f_avg3 = mean(1./(2*diff(intercept_times3)));
data1.f_avg = f_avg1; data3.f_avg = f_avg3;

%Call prep function
Ncycles = 6;
    %Number of line cycles over which to perform analysis
data1.Ncycles = Ncycles; data3.Ncycles = Ncycles;

if 1%data1.time(1) > 1.504187496000001e+15
    figure();
    plot(unix2matlab_EST(time), V1, 'r');
datetickzoom;
endif

    %
    %
figure();
    plot(unix2matlab_EST(time), V3, 'k');
datetickzoom;

    %
close all;

data1 = Pete_prep(data1,1);
data3 = Pete_prep(data3,1); %Run Pete_prep

S1(ind:ind+length(data1.S)-1,:) = data1.S; S3
(ind:ind+length(data3.S)-1,:) = data3.S;
fl(ind:ind+length(data1.f)-1,:) = data1.f; f3
(ind:ind+length(data3.f)-1,:) = data3.f;
tvect1(ind:ind+length(data1.tvect)-1,:) =
data1.tvect; tvect3(ind:ind+length(data3.
tvect)-1,:) = data3.tvect;
Lvect1(ind:ind+length(data1.Lvect)-1,:) =
data1.Lvect; Lvect3(ind:ind+length(data3.
Lvect)-1,:) = data3.Lvect;
Fsvect1(ind:ind+length(data1.Fsvect)-1,:) =
data1.Fsvect; Fsvect3(ind:ind+length(data3.
Fsvect)-1,:) = data3.Fsvect;

end

ind = ind + length(data1.S);

end

data1.S = S1; data1.f = fl; data1.tvect = tvect1; data1.
Lvect = Lvect1; data1.Fsvect = Fsvect1;
data3.S = S3; data3.f = f3; data3.tvect = tvect3; data3.
Lvect = Lvect3; data3.Fsvect = Fsvect3;

alrt = load('handel');
sound(alrt.y, alrt.Fs)

B.6.2 pete_prep.m

This script is what main_analysis_tabular.m calls to analyze the frequency and harmonic amplitudes of a voltage data segment.
clear;
close all;

Fs = 8000;

% Specify Dataset Filename
filename = 'C:\Users\2016\Dropbox (MIT)\Tests\AUG 31 2017\MIT Log\NILM\iv.txt';

filename = 'C:\Users\2016\Dropbox (MIT)\Tests\SEP 12 2017\MIT Log\NILM Data\iv.txt';

filename = 'C:\Users\2016\Dropbox (MIT)\Tests\SEP 14 2017\MIT Log\NILM Data\iv.txt';

filename = 'C:\Users\2016\Dropbox (MIT)\Tests\SEP 26 2017\Contact Data\iv.txt';

filename = {'F:\31AUG17\iv.txt', ...

'F:\12SEP17\_iv1.txt', 'F:\12SEP17\_iv2.txt', ...

'F:\14SEP17\_iv1.txt', 'F:\14SEP17\_iv2.txt', ...

'F:\26SEP17\_iv1.txt', 'F:\26SEP17\_iv2.txt'};

filename = {'E:\21NOV17\_iv1.txt', 'E:\21NOV17\_iv2.txt'

'E:\26SEP17\_iv1.txt', 'F:\26SEP17\_iv2.txt'};

for q = 1:length(filename)

% Get Data

test_td = tabularTextDatastore(filename{q});
test_td.ReadSize = Fs*30; %Read 30s of data at a time

% Run Pete_prep on tabular data store incrementally
% Measure size of tabular data file
len = count_rows(test_td);

disp('done counting rows!');

% Conservatively (oversize) preallocate data arrays
S1 = -1e3*ones(round(2.5*len/Fs*60),8); S3 = S1;
f1 = -1e3*ones(round(2.5*len/Fs*60),1); f3 = f1;
tvect1 = -1e3*ones(round(2.5*len/Fs*60),1); tvect3 = tvect1;
Lvect1 = -1e3*ones(round(2.5*len/Fs*60),1); Lvect3 = Lvect1;
Fsvect1 = -1e3*ones(round(2.5*len/Fs*60),1); Fsvect3 = Fsvect1;

ind = 1;
while hasdata(test_td)
    % Extract data from tabular file
    test = table2array(read(test_td));
    time = test(:,1);
    V1 = test(:,3);
    V3 = test(:,2);

    disp([' reading from ', datestr(unix2matlab_EST(time(1))), ' to ', datestr(unix2matlab_EST(time(end)))])
    run = 1;
    if unix2matlab_EST(time(1)) > 1.505225463012702e+15 || unix2matlab_EST(time(end)) < 1.505225463012702e+15
run = 0;

\textbf{end}

\textbf{if} run $= 1$

\begin{verbatim}
%Create data structure for passing test data to functions
data1.Fs = Fs; data3.Fs = Fs;
data1.time = time; data3.time = time;
data1.Va = V1; data3.Va = V3;
\end{verbatim}

\begin{verbatim}
%Get zero-intercepts of Va and calculate average frequency

data1 = Zero_Interceptor_Times(data1);
data3 = Zero_Interceptor_Times(data3);
\end{verbatim}

\begin{verbatim}
%Allocate to local variable for analysis
intercept_times1 = data1.intercept_times;
intercept_times3 = data3.intercept_times;
intercept_directions1 = data1.intercept_directions;
intercept_directions3 = data3.intercept_directions;
\end{verbatim}

\begin{verbatim}
f_avg1 = mean(1./(2*diff(intercept_times1)));
f_avg3 = mean(1./(2*diff(intercept_times3)));
data1.f_avg = f_avg1; data3.f_avg = f_avg3;
\end{verbatim}

\begin{verbatim}
%Call prep function
Ncycles = 6; %Number of line cycles
\end{verbatim}
over which to perform analysis

data1.Ncycles = Ncycles; data3.Ncycles = Ncycles;

if 1%datal.time(1) > 1.504187496000001e+15
    figure();
    plot(unix2matlab_EST(time), V1, 'r');
    datetickzoom;
    figure();
    plot(unix2matlab_EST(time), V3, 'k');
    datetickzoom;
    close all;

data1 = Pete_prep(data1,1);
data3 = Pete_prep(data3,1); %Run Pete_prep

S1(ind:ind+length(data1.S)-1,:) = data1.S; S3
  (ind:ind+length(data3.S)-1,:) = data3.S;
f1(ind:ind+length(data1.f)-1,:) = data1.f; f3
  (ind:ind+length(data3.f)-1,:) = data3.f;
tvect1(ind:ind+length(data1.tvect)-1,:) =
  data1.tvect; tvect3(ind:ind+length(data3.
  tvect)-1,:) = data3.tvect;
Lvect1(ind:ind+length(data1.Lvect)-1,:) =
  data1.Lvect; Lvect3(ind:ind+length(data3.
  Lvect)-1,:) = data3.Lvect;
Fs vect1(ind:ind+length(data1.Fs vect)-1,:) =
  data1.Fs vect; Fs vect3(ind:ind+length(data3.
  Fs vect)-1,:) = data3.Fs vect;
ind = ind + length(data1.S);
end
B.6.3 Zero_Intercept_Times.m

This script detects zero crossings in the most simple way possible: by looking for times at which the signal transits from negative to positive or vice versa. It works well for relatively clean sinusoidal signals, but should be modified for use with noisier signals or sinusoids with larger harmonic distortion.

```matlab
function out = Zero_Interceptor_Times(data)
% this function assumes a relatively well behaved periodic
% function like a
% sine curve, monotonic data near zero crossings. It returns
% the linearly
% interpolated time of the zero crossing.

out = data;
clear data;

% Allocate variables
timeVector = out.time;
sig = out.Va;
```
% Perform analysis

% timeVector = timeVector(:);

dd = (sig > 0);

e = diff(dd);

gg = (ee ~= 0);

cross_idx = find(gg == 1);

pair_ind = [cross_idx cross_idx+1];

intercept_times = (-sig(pair_ind(:,1)).*(timeVector(pair_ind(:,2)) - timeVector(pair_ind(:,1)))) ./
                     (sig(pair_ind(:,2)) - sig(pair_ind(:,1))) + timeVector(pair_ind(:,1));

intercept_directions = sign(sig(pair_ind(:,2)) - sig(pair_ind(:,1)));

if sig(pair_ind(1,1)) < 0
    intercept_times(1) = [];
    intercept_directions(1) = [];
end

% Allocate new variables to output structure

out.intercept_times = intercept_times;
out.intercept_directions = intercept_directions;

% plot(unix2matlab_EST(timeVector), sig); datetickzoom;
% disp(num2str(mean(diff(intercept_times*1e-6))));
% disp(num2str(max(diff(intercept_times*1e-6))));
B.6.4 view_trans.m

This script allows recorded transients that are logged in the file transient_log.xlsx to be viewed together and organized into figures based on the event that caused them and the base load at which they occurred.

```matlab
function view_trans(ind, streams, use_category)
%view_trans(ind, streams, use_category(0: by ID, 1: by type))
%Struct to organize data for plotting
clf; close all;
home = pwd;

figs = struct();

%Load transient guidetext for figure assignment
[num,~,raw] = xlsread('Tests\Transient_Log.xlsx');

%If indexing transients by type, recast ind accordingly
ind_counter = 1;
if use_category == 1
    ind_new = zeros(1,length(num(:,1)));
    for i = 1:length(ind)
        match = num(num(:,11)==ind(i),1);
        ind_new(ind_counter:ind_counter + length(match)-1) = match;
end
```
\[
\text{ind\_counter} = \text{ind\_counter} + \text{length}(\text{match});
\]

end

\[
\text{ind} = \text{ind\_new}(\text{ind\_new} \sim 0);
\]

end

% If \text{ind} = -1, plot ALL transients
if \text{ind} = -1
\[
\text{ind} = 1: \text{length}(\text{num});
\]
end

% Variable \text{ind} now represents all transients to be plotted.
% The next step is
% to organize the transients into groups by transient type
for \text{i} = 1: \text{length}(\text{ind})
% Load a transient
load(['../\text{Transients/trans}',\text{num2str}(\text{ind}(i))]);

% Add the transient to the figure bank
\text{if num(ind(i),11) \sim 0}
% If the figure of current transient is not represented
\text{if \sim ismember(['f',\text{num2str}(\text{num(ind(i),11))}],\text{fieldnames}(\text{figs}))}
% Create a new figure field
\text{figs.(['f',\text{num2str}(\text{num(ind(i),11))])] = struct('ind',1,'data',{cell(1,\text{length}(\text{ind}))},'ID',{cell(1,\text{length}(\text{ind}))},'Legend',{cell(1,\text{length}(\text{ind}))},'title',[],'type',\text{num(ind(i),11))});
end
% Insert transient at the end of the figure field, increment field index

```matlab
figs{['f',num2str(num(ind(i),11))}.data{figs{['f',num2str(num(ind(i),11))}.ind} = transient;
figs{['f',num2str(num(ind(i),11))}.ID{figs{['f',num2str(num(ind(i),11))}.ind} = ind(i);
figs{['f',num2str(num(ind(i),11))}.Legend{figs{['f',num2str(num(ind(i),11))}.ind} = num2str(ind(i))
```

cd('..:/Steady-State Recognition Tools');
[pwr,~,~] = est_pwr(raw(ind(i)+1,:));

cd(home);

gen_str = print_Gen_state(raw{ind(i)+1,8});
[ECU_str,~,~] = print_ECU_states(raw{ind(i)+1,2:5});
[LB_str,~,~] = print_LB_state(raw{ind(i)+1,7});
if strcmpi(gen_str(end-1:end), 'on')
    num_gen = 1;
elseif strcmpi(gen_str(end-2:end), 'off')
    num_gen = 2;
else
    num_gen = str2double(gen_str(1));
end
```
figs([f',num2str(num(ind(i),11))]).type = num(ind(i),11);
figs([f',num2str(num(ind(i),11))]).ind = figs([f',num2str(num(ind(i),11))]).ind + 1;
end
end

%Eliminate empty fields from each figure bank element
fn = fieldnames(figs);
for i = 1:length(fieldnames(figs))
    figs(fn{i}).data = figs(fn{i}).data(1:figs(fn{i}).ind-1);
    figs(fn{i}).ID = figs(fn{i}).ID(1:figs(fn{i}).ind-1);
    figs(fn{i}).Legend = figs(fn{i}).Legend(1:figs(fn{i}).ind-1);
end

%Now we have all requested transients organized into the struct "figs." The
%fields of "figs" are named 'fXXX,' where XXX represents the figure
%number and, equivalently, the transient type. Next, we plot them.

cd('../Transient Recognition Tools');

%TODO:
fn = fieldnames(figs);
for q = 1:length(streams)
    for i = 1:length(fn)
        fig = figure(figs.(fn{i}).type+100*(q-1)); hold all;
        for s = 1:length(figs.(fn{i}).data)
            sig = medfilt1(figs.(fn{i}).data{s}.(streams{q})),1,'truncate');
            sig = sig - find_base(sig);
            plot(figs.(fn{i}).data{s}.t-figs.(fn{i}).data{s}.t(1), sig);
        end
        plot(figs.(fn{i}).data{s}.t-figs.(fn{i}).data{s}.t(1), medfilt1(figs.(fn{i}).data{s}.(streams{q}))-mean(figs.(fn{i}).data{s}.(streams{q})),250,'truncate'));
        plot(superLPF(figs.(fn{i}).data{s}.(streams{q}))-mean(figs.(fn{i}).data{s}.(streams{q})),pi/500));
        title({{streamsjq},figs.(fn{i}).title});
        datetickzoom;
        legend(figs.(fn{i}).Legend);
    end
end
cd(home);

B.6.5 anlz_3.m

This is the script used to evaluate and fit a behavioral model to the seventh harmonic measured at the BCIL.

function [g1l, g1h, g3l, g3h, gbl, gbh] = anlz3(dtype, streams)
close all;
il = struct('P7',1,'Q7',1,'S7',1); fl = struct('P7',4,'Q7',5,'S7',6);

i3 = struct('P7',1,'Q7',1,'S7',1); f3 = struct('P7',4,'Q7',5,'S7',6);

ib = struct('P7',1,'Q7',1,'S7',1); fb = struct('P7',7,'Q7',8,'S7',9);

%31aug = r, 12sep = g, 14sep = b, 26sep = m, 21nov = k

colors_LB = {'g', 'm', 'b', 'm', 'b', 'm'};

colors_ECU = {'b', 'r', 'b', 'b', 'k', 'k', 'k', 'k', 'k', 'k', 'k', 'k'};

colors_ECU = {'k', 'k', 'k', 'k', 'k', 'k', 'k', 'k'};

colors_ECU = {'k', 'k', 'k', 'k', 'k', 'k', 'k', 'k'};

if strcmpi(dtype, 'lb')
    fnames = fnames_LB;
end
colors = colors_LB;
else if strcmpi(dtype, 'ecu')
    fnames = fnames_ECU;
    colors = colors_ECU;
else if strcmpi(dtype, 'load')
    fnames = [fnames_LB, fnames_ECU];
    colors = [colors_LB, colors_ECU];
end

gll = struct(); g2l = struct();
g3l = struct(); g3h = struct();
gb1 = struct(); gbh = struct();

for i = 1:length(streams)
    for s = 1:length(fnames)
        % Read load and indices
        [num, txt, raw] = xlsread(['..\..\..\..\..\..\..\Tests\Generator IV Characteristic\', fnames{s}]);
        % Read prep data
        date = datestr(datemax(txt{1,1})); % Test date
        testnum = txt{1,2} end; % Test number
        if ~isempty(strfind(txt{1,3}, '(1)')) || ~isempty(strfind(txt{1,3}, '(3)'))
            capacity = .60;
        else
            capacity = 1.2;
        end
if strcmpi(date,'31-aug-2017')
    date = 'AUG31';
    %Load AUG31 data
    d1 = load('C:\Users\2016\Dropbox (MIT)\BCIL\Code\MATLAB\BCIL Test Data Analysis\2017 Summer\Pete\31aug_ECU1.mat');
    d1 = d1.event1;
    d3 = load('C:\Users\2016\Dropbox (MIT)\BCIL\Code\MATLAB\BCIL Test Data Analysis\2017 Summer\Pete\31aug_ECU3.mat');
    d3 = d3.event3;
elseif strcmpi(date,'12-sep-2017')
    date = 'SEP12';
    %Load SEP12 data
    d1 = load('C:\Users\2016\Dropbox (MIT)\BCIL\Code\MATLAB\BCIL Test Data Analysis\2017 Summer\Pete\12sep_ECU1.mat');
    d1 = d1.event1;
    d3 = load('C:\Users\2016\Dropbox (MIT)\BCIL\Code\MATLAB\BCIL Test Data Analysis\2017 Summer\Pete\12sep_ECU3.mat');
    d3 = d3.event3;
elseif strcmpi(date,'14-sep-2017')
    date = 'SEP14';
    %Load SEP14 data
    d1 = load('C:\Users\2016\Dropbox (MIT)\BCIL\Code\MATLAB\BCIL Test Data Analysis\2017 Summer\Pete\14sep_ECU1.mat');
    d1 = d1.event1;
d3 = load('C:\Users\2016\Dropbox (MIT)\BCIL\Code\MATLAB\BCIL Test Data Analysis\2017 Summer\Pete\14sep_ECU3.mat');

d3 = d3.event3;

elseif strcmpi(date, '26-sep-2017')
    date = 'SEP26';
    %Load SEP26 data
    d1 = load('C:\Users\2016\Dropbox (MIT)\BCIL\Code\MATLAB\BCIL Test Data Analysis\2017 Summer\Pete\26sep_ECU1.mat');
    d1 = d1.event1;
    d3 = load('C:\Users\2016\Dropbox (MIT)\BCIL\Code\MATLAB\BCIL Test Data Analysis\2017 Summer\Pete\26sep_ECU3.mat');
    d3 = d3.event3;

elseif strcmpi(date, '21-nov-2017')
    date = 'NOV21';
    %Load NOV21 data
    d1 = load('C:\Users\2016\Dropbox (MIT)\BCIL\Code\MATLAB\BCIL Test Data Analysis\2017 Summer\Pete\21nov_ECU1.mat');
    d1 = d1.event1;
    d3 = load('C:\Users\2016\Dropbox (MIT)\BCIL\Code\MATLAB\BCIL Test Data Analysis\2017 Summer\Pete\21nov_ECU3.mat');
    d3 = d3.event3;

end

if ~isfield(gll, date)
    gll.(date).(streams{i}) = []
end
elseif ~isfield(g1l.(date), streams{i})
g1l.(date).streams{i} = []; glh.(date).streams{i} = [];
end

if ~isfield(g3l, date);
g3l.(date).streams{i} = []; g3h.(date).streams{i} = [];
elseif ~isfield(g3l.(date), streams{i})
g3l.(date).streams{i} = []; g3h.(date).streams{i} = [];
end

if ~isfield(gbl, date);
gbl.(date).streams{i} = []; gbh.(date).streams{i} = [];
elseif ~isfield(gbl.(date), streams{i})
gbl.(date).streams{i} = []; gbh.(date).streams{i} = [];
end

%% Process data
%Load and index info.
load_vec = num(:,1)/capacity;
st_vec = num(:,2);
nd_vec = num(:,3);

%Absolute value (V) v. base load
harm_vec_abs1 = zeros(length(load_vec),1);
harm_vec_abs3 = harm_vec_abs1;

for q = 1:length(harm_vec_abs1)
    harm_vec_abs1(q) = mean(d1.(streams{i}))(st_vec(q) : nd_vec(q));
    harm_vec_abs3(q) = mean(d3.(streams{i}))(st_vec(q) : nd_vec(q));
end

% dV/dLoad v. base load
harm_vec_diff1 = diff(harm_vec_abs1);
harm_vec_diff3 = diff(harm_vec_abs3);

load_vec_diff = diff(load_vec);
base_load_vec = load_vec(1:end-1);
clr = {'k', 'b', 'r', 'g'}; siz = {20 24 26 28};

% Plot data
if isempty(strfind(txt{1,3}, '(1)'))
    ch = '*';
    g1l.(date).(streams{i}) = [g1l.(date).(streams{i}) load_vec];
    g1h.(date).(streams{i}) = [g1h.(date).(streams{i}) (harm_vec_abs1+harm_vec_abs3)/2];
if strcmpi(date, 'NOV21')
    figure(fl.(streams{i})); hold on;
    scatter(load_vec, (harm_vec_abs1+
        harm_vec_abs3)/2, siz{i1.(streams{i})},
        clr{i1.(streams{i})});
    i1.(streams{i}) = i1.(streams{i}) + 1;
end
elseif ~isempty(strfind(txt{1,3},'(3')))
    ch = 'x';
    g3l.(date).(streams{i}) = [g3l.(date).(streams{i})]; load_vec];
    g3h.(date).(streams{i}) = [g3h.(date).(streams{i})]; (harm_vec_abs1+harm_vec_abs3)/2];
    if strcmpi(date,'NOV21')
        figure(f3.(streams{i})); hold on;
        scatter(load_vec, (harm_vec_abs1+ 
            harm_vec_abs3)/2, siz{i3.(streams{i})},
            clr{i3.(streams{i})});
        i3.(streams{i}) = i3.(streams{i})+1;
    end
elseif ~isempty(strfind(txt{1,3},'(1&3')))
    ch = 'o';
    gbl.(date).(streams{i}) = [gbl.(date).(streams{i})]; load_vec];
    gbh.(date).(streams{i}) = [gbh.(date).(streams{i})]; (harm_vec_abs1+harm_vec_abs3)/2];
    if strcmpi(date,'NOV21')
        figure(fb.(streams{i})); hold on;
        scatter(load_vec, (harm_vec_abs1+ 
            harm_vec_abs3)/2, siz{ib.(streams{i})},
            clr{ib.(streams{i})});
        ib.(streams{i}) = ib.(streams{i})+1;
    end
else
    disp('you messed up!');
end
figure(i*2-1); hold all;
scatter(load_vec, harm_vec_abs1, [colors{s}, ch]);
scatter(load_vec, harm_vec_abs3, [colors{s}, ch]);
title([streams{i},' vs. Load']); xlabel('Load (% Capacity)'); ylabel('Amplitude (V)');

figure(i*2); hold all;
scatter(base_load_vec, harm_vec_diff1./
load_vec_diff, [colors{s}, ch]);
scatter(base_load_vec, harm_vec_diff3./
load_vec_diff, [colors{s}, ch]);
title(['^d',streams{i},'/{dLoad} vs. Load']);
xlabel('Load (% Capacity)'); ylabel(['^d',streams{i},'/{dLoad}']);
end

end

figure(1);
% h(1) = scatter(nan,nan,'k*'); h(2) = scatter(nan,nan,'ko');
h(3) = scatter(nan,nan,'b*'); h(4) = scatter(nan,nan,'bo ');
legend(h,'21NOV (1Gen)', '21NOV (2Gen)', '14SEP (1Gen)',
'14SEP (2Gen)');
figure(3);
% k(1) = scatter(nan,nan,'k*'); k(2) = scatter(nan,nan,'ko');
k(3) = scatter(nan,nan,'b*'); k(4) = scatter(nan,nan,'bo ');
legend(k,'21NOV (1Gen)', '21NOV (2Gen)', '14SEP (1Gen)',
'14SEP (2Gen)');
B.6.6 test_identifier.m

This script is used to validate a transient identification algorithm using real data gathered at the BCIL. It takes all of the transients ever recorded, forms an exemplar library from them, and then forms a reduced exemplar library by removing some of the transients from each index of the exemplar library that is populated by more than one transient. This allows cross-validation by using the reduced exemplar library to identify every transient in the set.

```matlab
try
    id.delete();
gen.delete();
catch
    0;
end
clear all;
close all;
clear classes;

id = identifier(4);
id.init('test', '', 0.0083, 0, 0)
id.exemplar_clear(5);

%Load transient reference base
[num,~,raw] = xlsread('C:\Users\2016\Dropbox (MIT)\BCIL\Tests\Transient_Log.xlsx');
ds = load('../../Transients/trans',num2str(i)); ds = ds.
event1;
gen = gen(ds);
for i = 103:629
    load([ '../../ Transients/trans',num2str(i)]);
```
%Measure power
[pwr, eclass] = est_pwr(raw(i+1,:));
[num_gen, gclass] = print_Gen_state(raw{i+1,8});
class = eclass+gclass;

load_rel = 100*pwr/(id.C*num_gen);
%Feed transient frequency data into exemplar manager
id.exemplar_update(load_rel, class, num_gen, transient);
end

load( './Exemplar_Library/current_library', 'library');
save( './Exemplar_Library/saved_library', 'library');
library = id.exemplars;
save( './Exemplar_Library/current_library.mat', 'library');

id.exemplar_randomize();

% st_inds = [596700, 718350, 838001, 974000, 1106001, 1226000, 1359000, 1215000, 1467001, 1098000];
% nd_inds = [715600, 838000, 963300, 1106000, 1213500, 1344900, 1467000, 1225000, 1485300, 1110000];
% init_gen = [1, 1, 2, 2, 1, 2, 1, 1, 1, 2];
% base_load = [0, 15, 0, 25, 15, 25, 15, 25, 15, 0];
% W = [1500, 1500, 1500, 1500, 1500, 1500, 1500, 1500, 1500, 1000];
% num_ECU = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
46
0, 4, 4, 0];

47 % st inds = [735500];
48 % nd inds = [838000];
49 % init gen = [2];
50 % base load = [15];
51 % W = [2400];
52 % num ECU = [0];
53 % adj = [0];
54%

55 % DELETE ECU RUNS AND GENERATOR EVENTS
56 % st inds = [1098000, 1616600, 1467001, 1356000, 1345000,
1225000, 1215000, 110601, 970000, 956000, 838001,
852500, 848800, 596700, 725627, 715227];
57 % nd inds = [1110000, 1700600, 1485300, 1467000, 1359900,
1344900, 1225000, 1213500, 1106000, 970000, 951500,
951800, 852500, 715600, 838000, 725627];
58 % init gen = [2, 2, 1, 1, 2, 2, 1, 2, 1, 2, 1];
59 % base load = [0, 0, 15, 15, 15, 25, 25, 15, 25, 25, 0, 0, 0, 0, 15, 15];
61 % num ECU = [0, 0, 4, 0, 0, 0, 4, 0, 0, 4, 0, 0, 0, 4];
%% DETECT GENERATOR EVENTS (does not detect entry #13)

\texttt{st\_inds} = [715227, 956000, 1215000, 1467001, 1900000, 1959000, 2043000, 849000, 1097000, 1350000, 1580000, 1950000, 2008000, 2100000];

\texttt{nd\_inds} = [725627, 970000, 1225000, 1485300, 1910000, 1964000, 2050000, 860000, 1110000, 1360000, 1600000, 1960000, 2016000, 2200000];

\texttt{init\_gen} = [1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2];

\texttt{base\_load} = [15, 25, 25, 15, 30, 30, 30, 0, 0, 15, 15, 0, 0, 0];

\texttt{W} = [1800, 1800, 1800, 1800, 1800, 1800, 1800, 1800, 1800, 1800, 1800, 1800, 1800];

\texttt{num\_ECU} = [4, 4, 4, 4, 3, 3, 3, 0, 0, 0, 0, 0, 0];

%%

\texttt{st\_inds} = [849000, 1097000, 1350000, 1580000, 1950000, 2008000, 2100000];

\texttt{nd\_inds} = [860000, 1110000, 1360000, 1600000, 1960000, 2016000, 2200000];

\texttt{init\_gen} = [2, 2, 2, 2, 2, 2, 2];

\texttt{base\_load} = [0, 0, 15, 15, 0, 0, 0];

482
77 \% W = [1800, 1800, 1800, 1800, 1800, 1800, 1800];
78 \% num_ECU = [0, 0, 0, 0, 3, 2, 2];

79

80 \% REJECT CONFEDERATE PUMPS
81 \% st_ind = [2121530];
82 \% nd_ind = [2143235];
83 \% init_gen = [2];
84 \% base_load = [0];
85 \% W = [1500];
86 \% num_ECU = [3];

87

88

89 \% st_ind = [1356000, 1106001];
90 \% nd_ind = [1467000, 1213500];
91 \% init_gen = [1, 1];
92 \% base_load = [15, 15];
93 \% W = [2400, 2400];
94 \% num_ECU = [0, 0];

95

96 \% st_ind = [725627];
97 \% nd_ind = [838000];
98 \% init_gen = [2];
99 \% base_load = [15];
100 \% W = [1200];
101 \% num_ECU = [0];

102

103 \% st_ind = [596700];
104 \% nd_ind = [715600];
% init_gen = [1];
% base_load = [0];
% W = [1500];
% num_ECU = [0];

% st_inds = [1467001, 1215000, 956000, 715227];
% nd_inds = [1485300, 1225000, 970000, 725627];
% init_gen = [1, 1, 1, 1];
% base_load = [15, 25, 25, 15];
% W = [2400, 2400, 2400, 2400];
% num_ECU = [4, 4, 4, 4];

LBnom = [0 5 10 15 20 25 30];
LB = [0, linspace(11.2, 64.9, 6)*3*120*le-3]/1.; %Estimate
    actual load bank loads (kW) for nominal
    (5,10,15,20,25,30)kW loads

result = zeros(length(ds.f), 2);
f = result;
S3 = result;
for i = 1:length(st_inds)
    disp('Next dataset!');
    disp(['num2str(init_gen(i)), ' Gens, ', num2str(base_load(i)), 'kW Base Load, ', num2str(num_ECU(i)), ' ECUs']);
    if i == 3
0;
end
id.Wlen = W(i)/120;
id.init('test', '', 1/120, 0, 0)
id.num_ecu = num_ECU(i);

484
id.num_gen = init_gen(i);

id.baseload = LB(find(LBnom==base_load(i)));

% [result{i}, f{i}, S3{i}] = track_load2(st_inds(i),
nd_inds(i), init_gen(i), base_id, num_ECU(i), ecu, genon,
genoff, W(i));

id.Gen = gen;

for s = st_inds(i):nd_inds(i)
    genindx = s;
    class = id.ID();
    result(s,1) = id.num_ecu(1);
    result(s,2) = id.num_gen(1);
end

end

%% Display test results
close all;
if ~isempty(result)
    figure();
    yyaxis left; plot(result(:,1)); hold on; plot(result(:,2)
    );
    ylabel('# ECUs/Gens. Active');
    yyaxis right; plot(ds.f);
    ylabel('Hz');
    legend('ECUs', 'Gens.', 'Line Freq.');
    ylim([58, 62]);
end

title(['Test Result', num2str(i)]);
Bibliography


[16] Long Ha, StÃ©phane Ploix, Eric Zamai, and Mireille Jacomino. Control of energy in home automation by resource constraint scheduling. 01 2005.


