Spring 2023

Solar Sonification: From Data to Music with Solar Protocol

Chris O. Stone , '23

Solar Sonification: From Data to Music with Solar Protocol

A Swarthmore College Engineering Design Project

Author: Chris Stone ‘23 (He/They)

Advisor: Professor Matt Zucker

2023-05-05

Table of Contents

1 Abstract

“Solar Sonification” is an interactive python notebook written in Jupyter Labs was designed to utilize data from a solar-powered web server to create and share data sonifications. It is an extension of the global network of solar-powered web servers known as Solar Protocol, one of which was built in 2020-2021 through the Halpern Family Foundation Engineering Design fund. Using this notebook, the participant is able to learn more about the functionalities of the server while engaging in their musical relationships being able to customize the pitches, instruments, and other musical parameters of the output as MIDI files. Files can be played using mediums such as Pygame within the Jupyter Notebook and external synthesizers connected to a device, or shared in audio formats including WAV and MP3. Future work can be done to improve the sonification process and musical aesthetics as an educational tool.

2 Acknowledgements

Thank you to Prof. Matt Zucker who advised me through this capstone project. When I approached you on whether I should do the fun thing or the boring but maybe more obviously practical thing, you encouraged me to do the fun thing, and that it would be just as much a practical thing. You were right and every day I've spent working on this project has felt worth it and made me a better person and engineer.

Thank you to Prof. Maggie Delano who served as my freshman year advisor and temporary capstone advisor. Since I first arrived at Swarthmore College, you have made me feel seen and included in the engineering department, not just as a student but as a person. In your classes, things actually make sense, and I feel empowered to move beyond the imposter syndrome that comes from not seeing people like me represented in this field.

Thank you to Prof. Carr Everbach, who took a chance on me when I least expected it and encouraged me to pursue the President's Sustainability Research Fellowship during the pandemic. In spite of the circumstances, it is because of you I developed a deep interest in energy as a topic that could take me beyond engineering, into cultural memory work, into food justice, and more.

Thank you to Cassy Burnett, Ed Jaoudi, and J Johnson, who made navigating the engineering department and making projects come to life accessible and enjoyable. You all are the duct tape keeping the department together.

3 Introduction

As the climate crisis worsens, renewable energy is facing a rapid demand, especially from governments facing scrutiny for their climate impact. However, information about solar energy technologies is largely inaccessible to the public, which impedes the public's ability to participate in decision-making processes around their energy infrastructure, whether in support or critique. Solar cannot reach adoption effective to mitigating climate change should its implementation fail to address the structural injustices of our energy systems that created in the first place, as shown by such circumstances as the U.S. utilities shutting off power 5.7 million times as their profits continued to thrive [1], or that 1 out of 5 Americans struggle to pay their energy bill at least once a year [2], [3].

The Solar Sonification project an interactive Python-based Jupyter Notebook for learning about and creating music utilizing historical data from a Solar Protocol solar-powered web server located in the Singer Hall Solar Lab. At its core, it is guided by attempting to address the following prompt for energy-centered design posed in Solar Protocol's 2022 paper for the "Computing Within Limits" conference:

> How can we better convey, communicate and visualize the often invisible energetic attributes of computational technologies in user interfaces and online experiences? There are opportunities for both designing implicit relationships of form and function where, for example, energy availability might influence the size of assets or resolution of media… The goal is to foster energy literacy at the point of design and engineering, where those in these fields are encouraged to be cognizant of the effects of their decisions. [4, p. 5]

Whereas the original prompt focuses on visibility, Solar Sonification centers on audibility, where many of the same opportunities can be found though for a different audience of energy literacy that engages the sense of hearing.

4 Background

4.1 Solar Protocol

Solar Protocol is a global network of solar-powered web servers for the website https://www.solarprotocol.net. Each web server runs on a Raspberry Pi, which copies data from the serial addresses of the charge controller approximately every 2 minutes, checks which server is acting as the current host for the Solar Protocol site, calls the scaled PV power data for all the other web servers, then directs the site's web traffic to

whichever web server at the time has the most PV power data [4, p. 3]. It was designed for the Eyebeam Rapid Response Fellowship in 2020 by Alex Nathanson, Tega Brain, and Benedetta Piantella (who served as my primary contact through this project) and since then has found participants from across:

- Peterborough, Canada
- New York City, USA
- [Swarthmore], USA
- Santiago, Chile
- Nairobi, Kenya
- Newcastle, Australia
- Alice Springs, Australia
- Amsterdam, Netherlands
- Beijing, China
- Kalinago Territory, Dominica
  [4, p. 4]

Content on the Solar Protocol website experiences near constant uptime by redistributing its load onto different energy sources provided by those in a community agreement to participate in this experiment, in a manner that might not otherwise be feasible for every web server at that site individually. A demonstration of how the active server moves between servers is commonly expressed through data visualizations, one in particular is central on the front page of the main solar protocol site that shows four

active servers at a time as shown in Figure 1. The creative solar-power computational technology project serves as a line of inquiry into how alternatives can be collaboratively imagined given the realistic constraints that differ in the context of each server steward, especially when designed around the behavior of and relationships with natural phenomena.

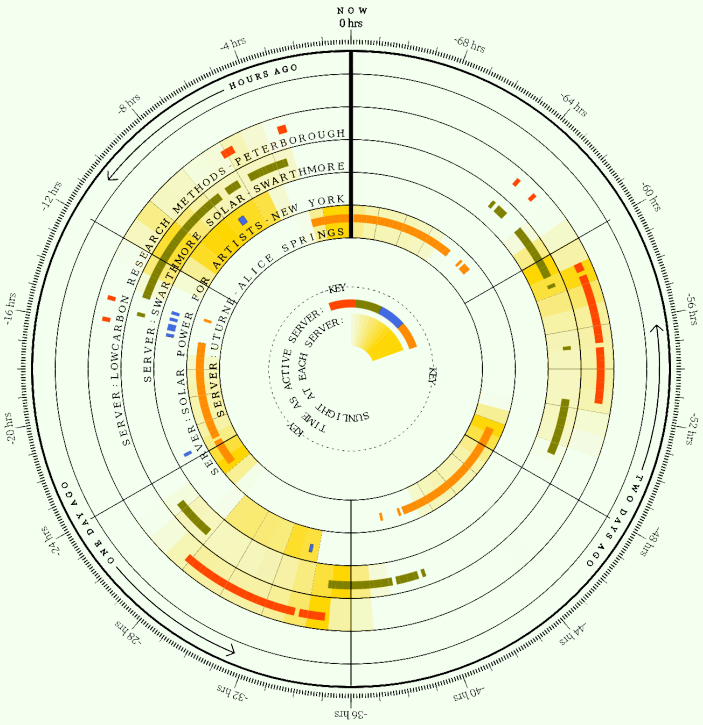


Figure 1. Visualization from https://www.solarprotocol.net showing 72 hours of network data.

4.1.1 System Functionality

The system has a common arrangement of charge controller connected to a panel, battery, and load at the load ports represented by Figure 2 which can be compared to

the connection illustration for a Solar Protocol web server in Figure 3 [5]. The load ports are similar to connecting a load to the battery and are limited to low power devices.

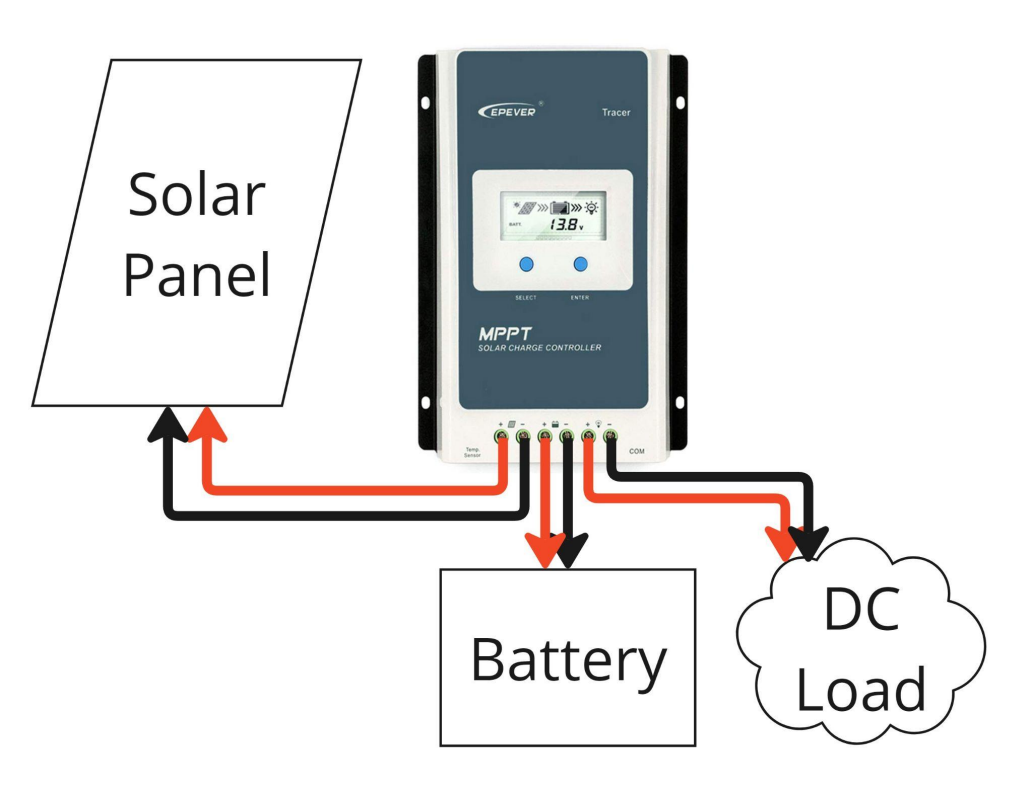


Figure 2. General diagram of connections to an MPPT charge controller to a solar panel, battery, and DC load.

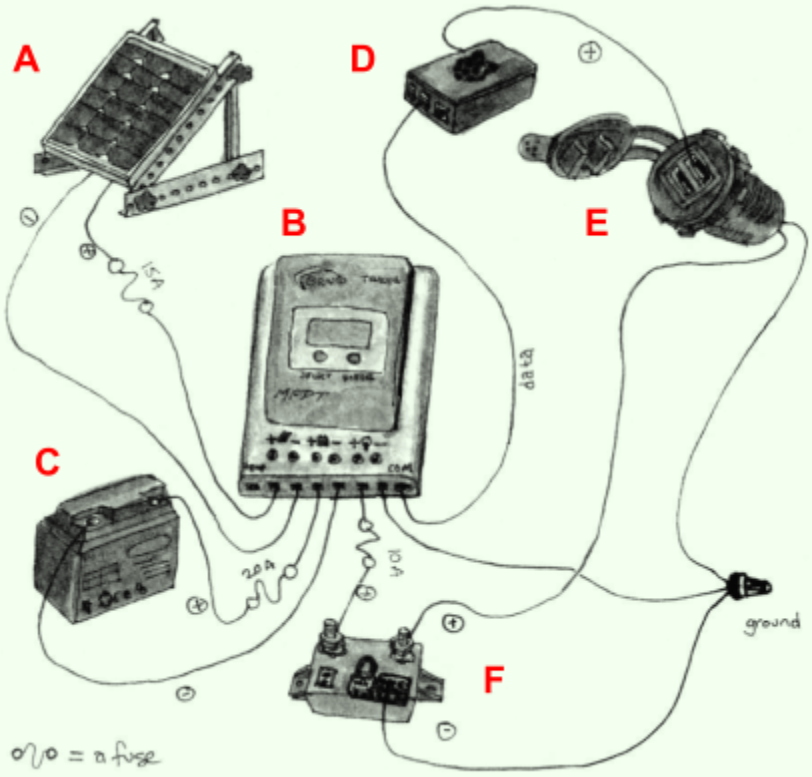


Figure 3. An annotated of the components for the Solar Protocol web server. Key components are: A. 50W 12V Monocrystalline PV module, B. Epever 20A solar MPPT charge controller, C. 12V 22Ah AGM sealed lead-acid battery with battery terminals, D. Raspberry Pi 4, E. 12V to 5V USB power outlet with an inline 10A fuse, F. Victron battery protection circuit.

Understanding the system functionality is necessary to understand why what data is available and the behavior of the trends it represents. Along with the date and time, 10 different measurements are copied from the charge controller serial addresses by the Raspberry Pi to be posted approximately every 2 minutes. The variables are as follows:

1. Solar Panel Power
2. Solar Panel Voltage

3. Solar Panel Current
4. Battery Power
5. Battery Voltage
6. Battery Current
7. Rasberry Pi Power
8. Rasberry Pi Voltage
9. Rasberry Pi Current
10. Battery Percentage

The solar panel generates power during the day, which is distributed to the battery through the charge controller to meet the battery's demand to be charged, especially after running the server overnight. The charge controller with adjust the current and voltage to achieve the solar panel's maximum power point, and the same power will be exchanged to the battery at a different current and voltage depending on the battery's limits and requirements. Through the charge controller, the Raspberry Pi is connected in parallel to the battery, thus leading to the same voltage.

If the battery is fully charged, it will not demand as much power except for keeping the server on. Demand is relative to when the website is being visited or when the Raspberry Pi is the active server, but changes are usually not significantly noticeable when graphed. This usually occurs during the day when the server has the most

sunlight, but spikes in demand could also occur during the night if there are no other active servers with power.

If the battery percentage has discharged too low (for the Swarthmore server, this is about 34% or lower), the Raspberry Pi is disconnected, thus stopping data logging. The system going down due to battery discharge usually occurs in the evening of the same day or the early hours of the next day while the sun still is not out, and recovers in the next day(s) while the sun is out once the battery has been charged up to 50% or more. Using my data processing program mentioned later in my implementation, I found no record for the Swarthmore Solar Protocol server of this happening multiple times a day. Other factors, such as updates to the server at the core Solar Protocol codebase, could also lead to pauses in the data logging leading to missing data not related to a deficiency of power.

Most of these aspects of the system I learned at the start of the capstone after completing the web server build and reviewing its data. Prior to doing so, I had not realized how intertwined the data were and how that would affect my approach to

sonification. For example, only sonifying the power measurements and referring to this as encompassing the overall system behavior would be an insufficient representation.

4.2 Swarthmore Solar Protocol

After participating in the President's Sustainability Research Fellowship in 2020-2021 per the recommendation of Prof. Carr Everbach, I became deeply interested in what it would mean to design energy systems in the context of energy justice. I was awarded $900 with the Halpern Family Foundation Engineering Design Fund for my proposal to build a Solar Protocol web server on Singer Hall's Solar Lab. I pursued this project with an interest in alternative energy relationships that acknowledge planetary limits and are centered around degrowth, as well as building experience with hands-on hardware assembly behind digital infrastructure. In contrast to the latter, most users never interact with the physical infrastructure behind the digital infrastructure they utilize.

Under Prof. Everbach's mentorship, I began the Fall of 2020 purchasing materials as specified in the Solar Protocol Bill of Materials and incorporated a feasibility study as my final project for ENGR 035. Solar Energy Systems. I continued to assemble the server with the assistance of Ed Jaoudi and J Johnson into Spring of 2021 along with

research into the Campbell CR1000X data logger for the Singer Hall Solar Lab as a part of my directed reading. With the assistance of Jesse Li ‘22, Swarthmore Solar Protocol came online on April 9th, 2022, its final assembly shown in Figure 4.

Figure 4. Left: Photo of the Swarthmore Solar Protocol server on the back rack of the Singer Hall Solar Lab, with the solar panel mounted an adjustable mount designed by J Jaoudi and the enclosure on an acrylic shelf below the rack also designed by Jaoudi. Right: A photo of the inside of the enclosure, notably the Raspbery Pi is located in the center mounted with zip-ties to a perforated board. Next to it on the right are the MPPT charge controller and battery.

4.3 Sonification

Creating open-access to data alone is not enough. While data may be available, if it cannot be interpreted, it accumulates without a purpose. Energetic attributes can be expressed through sensory media aside from visual representations to facilitate embodied relations with the nature of a dataset. The interaction design foundation defines affordances as relational to the user: “what a user can do with an object based on the user’s capabilities” [6]. Sonification provides the affordance of listening to the data created by the system, and is beneficial both to auditory learners and those who enjoy auditory experiences in general. Statistical learning and pattern recognition has been found to be improved in auditory over visual modalities [7]. Though the system is made of non-living parts, the interpretation of the system’s activity as the intertwining of living and non-living kinship is unveiled to the listener by exposing them to the manner in which the digital, even as a digital audio signal or web services more broadly, cannot exist without the physical infrastructure and energy flows.

5 Design Considerations

5.1 Requirements

There are two forms of design ethos that my project should apply. The first is to consider the application of the first prompt for Energy-Centered Design as mentioned in the introduction, which includes incorporating effective strategies for sonification such as:

- Ensuring the scope of the sonification can be explained with concision to a listener with limited prior knowledge. [7]
- The aesthetic utilization of “signal referent” sounds or “ear-icons” (as opposed to visual icons). [7], [8]

The second is Tim Murrary-Browne’s “Against Interaction Design Manifesto” where he poses the concept of “Negotiated Interactions” that belong to and empower the participant as much as they do to the designer and the technology [9]. Thus, I am choosing to expose the code as a part of this sonification process with features that focus on the range of potential sonification work rather than an emphasizing a single sonification approach without context.

5.2 Constraints

After using Jupyter Labs on the Python and R distribution known as Anaconda in ENGR 014 my sophomore year, I began to use Miniconda as a minimalistic alternative to Anaconda. While it is not necessary to run Jupyter Labs in Miniconda, by using the tools I am most familiar with, I run into a conflict in that Conda does not have all the packages that are available to Python through pip. It is not a recommended practice to combine these packages, so I will have to use a virtual environment to install them and run the code. This should not be an issue for anyone else running Jupyter Labs through Python.

5.2.1 Professional Codes and Standards

Professional codes and standards also serve as constraints. There are two main aspects that are relevant: open-source practices and the MIDI standard.

Solar Protocol's open-source implementation pre-determines several aspects to which I will have to work along with. Solar Protocol's hardware guidance establishes specific components to use and how they should be connected, and its codebase establishes how activity is triggered on the web server as well as the data logging process for

outputting the CSV files I utilize in my sonification. As I plan to release this code publicly as a contributor to Solar Protocol, I will need to prepare for my code to meet expectations of the open-source community such as ensuring that the source code is easily accessible and can be interpreted by others than myself. Solar Protocol also does not have a specific license attached to it for the time being, though if or when it does, this could impact what it means for my code to be part of such a project in the future.

MIDI is the primary standard for communication between devices for musical synthesis, and utilizing it will maximize interoperability of this project with other MIDI programs commonly used. However, the feature set of the sonification caters toward Western music theory, notation, and cultures. Steps toward addressing concerns for this bias are being developed in MIDI 2.0. Work in decolonizing the electronic music tools has been done by people like Jon Silpayamanate [10] and Khyam Allami [11]. Therefore, the Solar Sonification project is but a gesture toward understanding and expressing the relationship between Solar Protocol Stewards and their web servers as music, and cannot fully capture all possible musical relationships.

6 Implementation

6.1 Background Research

Since Fall of 2020 I have been accumulating a number of projects that combine ecologically-centered technology with artistic expression, such as the projects Cadu' "Wind Line" [12], Gottfried Haider's "Study for a Camera on a Plot of Land in the Desert" [13], and Zach Poff's "Pond Station" [14] (which especially influenced my interest in place-based sounds, along with a number of internet radio stations). Before knowing what the word sonification meant, I had an understanding of how data loggers worked through working in the Singer Hall Solar Lab and the Campbell Scientific CR1000X. A friend showed me libi rose's "autopoetic printer" [15] after I learned the first term from Marisa Parham [16]. I considered creating something that would continuously print sheet music for a music box, inspired by Bryan Braun's "Music Box Fun" [17].

As Prof. Zucker encouraged me to look further into how MIDI worked without any electronic music background, I discussed this project with alumni Peter Wu '22, who

holds a background in both music and computer science. He shared with me the YouTube tutorial “Sonification with Python - How to Turn Data Into Music w Matt Russo (Part 1)” by NASA’s astrophysicist, musician, and sonification specialist, Dr. Matt Russo [18]. His tutorial materials would become the basis for my project. I received his approval to redistribute snippets of his code with modifications to make it compatible with Solar Protocol’s data.

I continued to developed my research specific to sonification through the following approaches:

- Reading news/academic articles and blogs on the topic of sonification and its purpose and MIDI
- Trying out tutorials and following forums of people starting or with extensive experience in sonification
- Discussing the project with music experts including Devine Lu Linvega, Duncan Greere, Paul Batchelor, and Prof. Jon Kochavi

6.2 Data Access

The system is mounted outside in a stationary location requiring me to SSH or “secure shell” into the server to access the data. At some point through the summer, I became unable to SSH in. We later learned that this was because my public key had been

removed during an update of the core Solar Protocol codebase. Once I regained access, I could SSH in and use SCP or "secure copy" to access one of the data files at a time. Per Piantella's recommendation, I used the Filezilla client to back up all the files locally using FTP or "File Transfer Protocol". Every once in a while, I'll manually go back in and backup more files.

6.3 Data Processing

I wanted to represent downtime in the system as silence. I worked with Prof. Zucker to take the datetime string and parse them into an interpretable datetime format based on UNIX time:

```
# accepts a date string in solar protocol log format
# see https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior
# see https://en.wikipedia.org/wiki/Unix_time

format_str = '%Y-%m-%d %H:%M:%S.%f'

date_obj = datetime.strptime(row_dict["datetime"], format_str)

if midnight is None:

            midnight = datetime.combine(date_obj.date(), datetime.min.time())

            # get a floating point number of seconds since 1970-01-01
            #time_in_seconds = date_obj.timestamp()

# get a floating point number of seconds since midnight
time_in_seconds = (date_obj - midnight).total_seconds()
current_row.append(time_in_seconds)
```

Once the datetimes are in this format, the date can be used to subtract the total number of seconds represented by the datetime at any point of the day from the number of seconds at midnight for that day. By doing so, all the datetime values will exist in a range from 0 to 86,400 (the total seconds in a day), accounting for when the data starts, contains, or ends with missing data. For example, the datetime value might be 2023-01-05 17:57:04.056222 and this would become 52606.205717 seconds after midnight (based on 2023-01-05 in UNIX time). A number like 52606.205717 will allow me to utilize my arrays in value mapping as opposed to using a datetime string.

I created a data processing file that looks through all the files in my backup folder to keep track of the following to see how significant their impact would be on the sonifications:

- The number of days when the data log starts significantly past midnight and the time when that occurs
    - Sonification would start with silence. Without the time conversion, the track would start abruptly at a time that does not represent the start of the day.
- The number of days and number of times per day when the data log has data missing during the middle of the day and the time when those occur.
    - Sonification would have silence in the middle of the track

- The number of days when the data log ends significantly before midnight and the time when that occurs
    - Sonification would end with silence. Without the time conversion, the track would end abruptly at a time that does not represent the end of the day.

To find these times, I started by finding all the gaps in the data to see how long downtime could be. It was through this process I discovered that there were some negative differences: instead of time linearly moving forward at all times, there are some data points with earlier times. The wider gap between a much earlier time and the return to the real time data can lead to a false positive when looking for downtime in the middle of the day. A graphical visualization has a linear scale which would automatically sort this point, making it non-obvious to the viewer that it appears in an inconsistent place in the data log. Without having clear reasoning as to why this was occurring after talking to Piantella, I submitted a bug report to the Solar Protocol GitHub and skipped these data points for the interim. From there I specifically looked for gaps that had differences larger than the approximate 2 minutes that the data log should be recording if the Raspberry Pi is on.

6.4 Sonification

6.4.1 Initializing Data

Sonifications are created on a file by file basis. After importing the necessary libraries, the CSV file for the day is converted into a structured array. The structured array allows me to refer to variables mentioned in the first row, known as the header file, by their name, to grab all the data in the column associated with the variable. On Feb 12, 2023, Figures 5-8 compose the voltages, currents, powers, and battery percentage. They are represented with multiple plots on one graph primarily for the purposes of this paper, in practice I generally look at each individually for scales relative to the range of the variable, rather than the range of multiple variables. Furthermore, while this is a day that the battery fully charges and discharges, there is still variation between the days depending on the amount of solar availability and weather conditions where that might not happen, or where there are empty spots where the system went down.

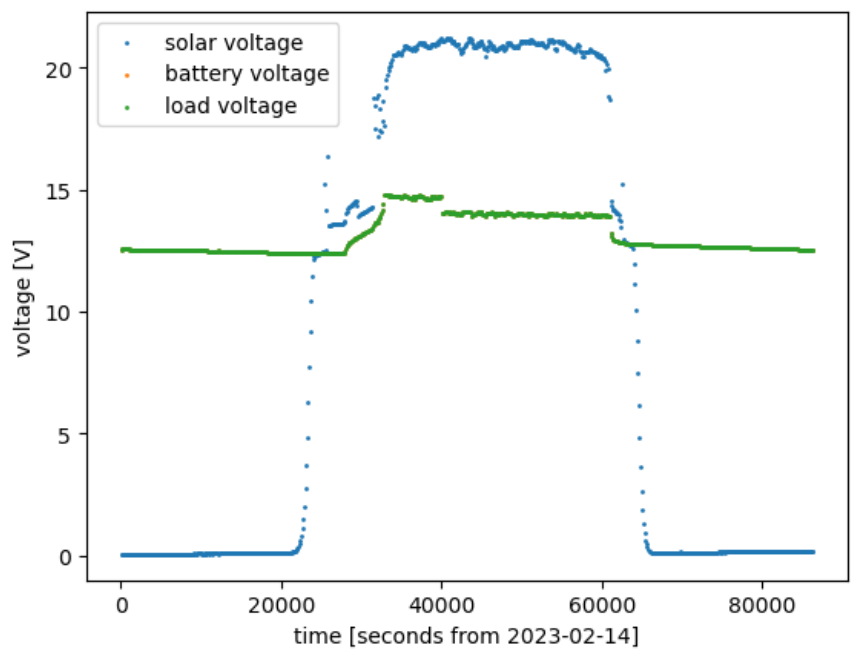


Figure 5. Voltages measured in volts on February 14, 2023. The battery voltage and load voltage overlap because they have the same value.

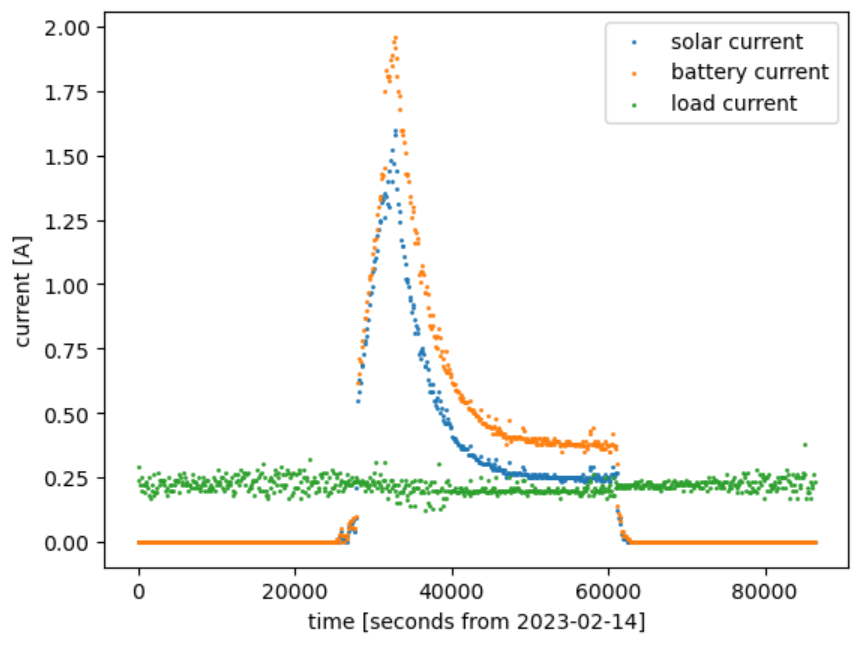


Figure 6. Currents measured in amps on February 14, 2023. The solar current and battery current tend to have similar trends, but they are not exactly the same.

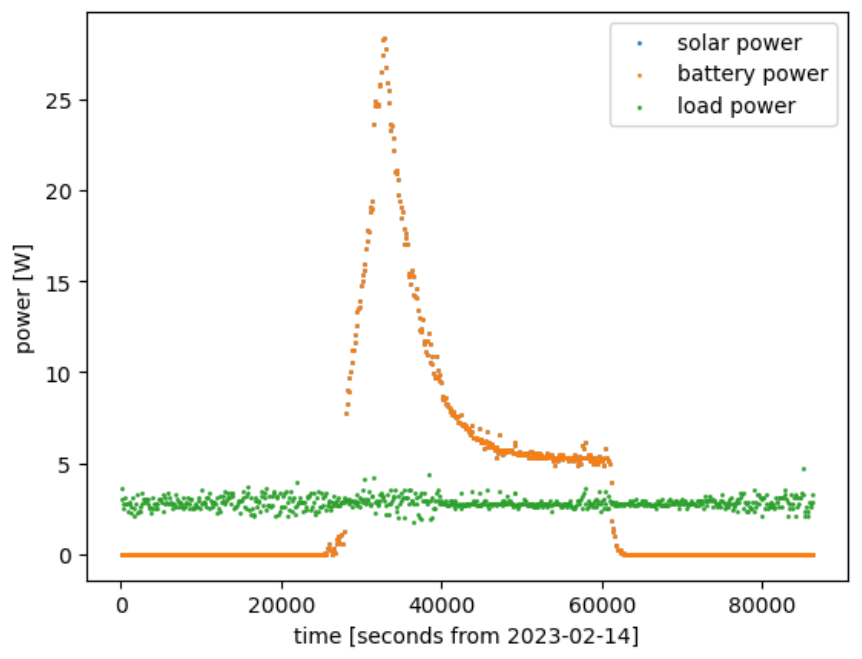


Figure 7. Power measured in watts on February 14, 2023. The solar panel power and battery power overlap because they have the same value.

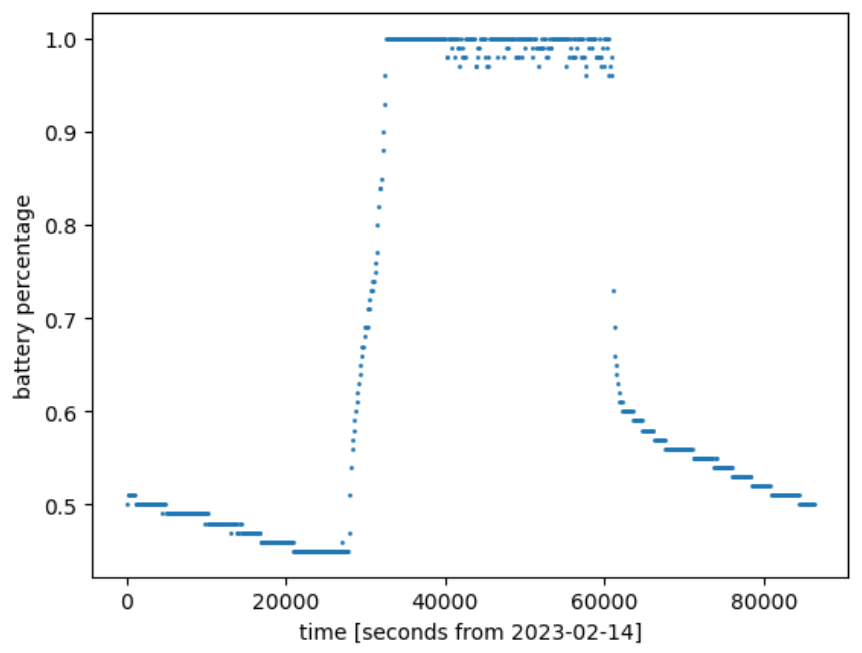


Figure 8. Battery percentage on February 14, 2023. The solar panel power and battery power overlap because they have the same value.

Once I have a sense of the trends of the data for the day, the aspects that define the story I want to tell, the variables are paired with relevant music parameters and transposed through a mapping process.

6.4.2 Mapping

Mapping is what lets me apply the data to a musical parameter. I've found linear mapping to be the most useful, represented by the snippet below, but other mapping algorithms should be possible if desired. The linear mapping takes values that exist within a range and maps them into resultants that exist in a new desired range:

```
result = min_result + ((value - min_value)/(max_value - min_value))*(max_result - min_result)
```

In some cases, it may be necessary to inverse the mapping if the desired musical parameter should have the opposite expression relative to the data. For the same variable, one could say, higher power should be a higher pitch because it is a high number, while another could say, higher power should be a lower pitch because a bassier note sounds more powerful.

6.4.2.1 Mapping to Time

When designing a sonification there is a tension between the ability to use sonification to share data at a pace that it is interpretable and the ability to share it within overall time and attention constraints [7]. For the purposes of the tutorial, I compress where time is spread out at 2 minute intervals that could make it harder to pay attention to changes in patterns for an untenable 24-hour listening time, down to an arbitrarily selected 3 minutes at a rate of 60 beats per second represented by the following snippet:

```
t_data = map_value(time, 0, 86400, 0, 180)
```

Map value sets the parameters of the value time, minimum and maximum value range, and minimum and maximum result range to produce the result of t_data. Three minutes felt personally acceptable while generating many sound files for the purposes of this project but could easily be modified. For variables that are zero until the sun is out, one can expect there to be more activity in the audio going centered into the first minute of the file until the second minute of the file, making it easier to pan through the file to find particular sections of the data. As future work considers how these sounds are incorporated into live performance, it is important to note the following:

> ...the data mapping can lead to absurd musical tasks for a human to perform. For instance, the latest [sonification using data from NASA's] Voyager 1 piece [on the flute] has 37 measures, and there is no place to rest or breathe until measure 32. [19]

Rounding or data cleaning processes could be used to reduce the number of data points utilized in a manner that would offer more consistency than the variations in the live data's sampling intervals.

6.4.3 Mapping to Other Musical Parameters

As mentioned earlier in the document, the sonification process is utilizing the MIDI standards which allows us to engage with several musical parameters by discrete numbers with a designated range for our resultant. When adding a note, the following parameters are available by the Python library MIDIUtil:

- track – The track to which the note is added.
- channel – the MIDI channel to assign to the note. [Integer, 0-15]
- pitch – the MIDI pitch number [Integer, 0-127].
- time – the time at which the note sounds. The value can be either quarter notes [Float], or ticks [Integer].
- duration – the duration of the note. Like the time argument, the value can be either quarter notes [Float], or ticks [Integer].
- volume – the volume (velocity) of the note. [Integer, 0-127].
- annotation – Arbitrary data to attach to the note.

[20]

A note is added for each row in the data to be played at the time series represented by the array `t_data` mentioned in the previous section which is passed through as floats for the default quarter notes. Pitch is the most commonly applied musical parameter in sonification. Matt Russo's sonification tutorial allows the participant to select their preferred starting note, scale intervals, and number of octaves to determine the array of pitches available to map the variable of interest onto. "Program changes" allows the participant to choose from 128 instruments such as violins (program 41) while "control changes" allow the participant to alter the dimensions of the note in the sense of timbre or depth effects. The following is an snippet of creating a track that plays the violin and uses the arrays `midi_data_1` and `t_data` to dictate the pitches and time after having been mapped to add notes to the track.

```
midiF = MIDIFile(1)  # one track
midiF.addTempo(track=0, time=0, tempo=bpm)
midiF.addProgramChange(0, 0, 0, 41)

for i in range(len(t_data)):
    midiF.addNote(
            track=0, channel=0, pitch=midi_data_1[i], time=t_data[i], duration=2,
volume=50
    )
```

A limitation to working on a per-file basis is that the range of values is determined by day, rather than overall. Therefore, the same value for a variable could be mapped to a different musical parameter result based on one day's data set compared to one another. Data processing could be used to find the absolute minimum and maximum values per variable based on all time data. A potential use would be to find how variables that reach higher values during peak sun hours may have a broader range of resultants in musical parameters in certain seasons than others.

6.4.4 Event-Based Actions

As noted in the System Function section, there are a number of particular events related to common system behavior that can be identified using the data. These are places to add "signal referent sounds" as mentioned earlier as effective sonification strategies. To start, there are three types of events I identify:

1. Times when the system goes down
    a. Specifically because the battery has discharged and disconnected the pi
    b. Or for any other unknown reason
2. Time when the system comes back online
    a. Specifically because the battery has recharged and connected the pi again
    b. Or for any other unknown reason
3. Times when the battery is fully charged

Each of these cases show how a time series can be created based on these events. For example use cases, I followed Steve Hein's "How to generate music with Python: The Basics" tutorial [21] to create rising and falling chord arpeggios as my "notification sounds". I apply a falling arpeggio to play when the system goes down because the battery discharged and a program change which changes the instrument from a piano to another instrument of choice every time the battery enters or leaves a "fully charged" range.

Times when the system goes down and comes back online during the middle of the day allows us to check sequential data points to see if it's related to the battery discharging to 34% and returning around 50%. A limitation to working on a per-file basis is that for downtime at the end of the day, you cannot validate the following data point without opening the following day's CSV file, and vice versa for downtime at the start of the day.

Data points that were marked with an unknown reason could be attributed to maintenance time, though due to their not being a clear record of all maintenance

periods to validate this, an issue was submitted to the Solar Protocol GitHub in case they raise concerns with others using the Solar Protocol dataset unfamiliar with the full scope of the system's behavior. One might imagine a sound of chaos or confusion to play when downtime is caused by an undetermined factor.

6.4.5 Playing and Exporting

Matt Russo's tutorial uses the python module set Pygame to play the MIDI file within Jupyter Labs for playing files immediately. It will only stop when a stop code is run and cannot be panned through. MIDI files are often not compatible with web audio players, preventing them from being easily shareable. To share sonifications easily, I explored more options to convert these files within the notebook.

MIDI files are usually converted to audio using a DAW or "Digital Audio Workstation" with a graphical interface like Ableton Live. While there are free DAWs available, mainstream DAWs tend to be expensive, resource intensive, and come with a large learning curve. With the intent that this could be code that runs off a pi, I followed Adam Dingle's "Playing music from Python via fluidsynth" tutorial [22]. FluidSynth is a software synthesizer that utilizes SoundFonts to take MIDI information to output

audio in the form of WAV files. Soundfonts are shared widely online but take up large file sizes that need to be managed with Git Large File storage. WAV files are also high quality sound files at sizes that are often larger than necessary for casual listening. The multimedia framework FFmpeg was used to convert the WAV files to MP3 files at more manageable sizes for redistribution, such as via social media. Improvements to this exporting process to reduce the need for both FluidSynth and FFmpeg would definitely be worth exploring.

6.4.6 Live Performance

My interest in incorporating these sonifications into live performances include both the practice of programming electronic music and visuals live as a creative performance is known as “live coding” and projects that connect sensors to plants and fungi and use synthesizers to play generated music. Using the Mido python library, participants can access external synthesizers connected to their device through their python file to play MIDI files similarly to Pygame. I tested this using the Arturia Microfreak, my first synthesizer. I learned that only one channel of MIDI notes can be played through the device at a time despite being a paraphonic device, namely on channel 0 and no others. Given this dataset has multiple variables that are being

recorded at the same time, if these are spread across multiple channels, a multitimbral synthesizer would be needed.

7 Conclusion and Future Work

In the Solar Sonification Project, I was able to successfully achieve my planned minimum viable product: a reproducible script that can process web server data, map data from Swarthmore Solar Protocol to musical parameters, implement effective sonification strategies, and produce shareable outputs that can be shared on the Swarthmore Solar Protocol Site. In the process of pursuing sonification, I learned several aspects about Solar Protocol function that I had not known from simply assembling the hardware, and identified points where the system could be improved or built upon through the lens of sonification. Considering my limited knowledge of formal music education, the process of being able to achieve a sonification effect following my existing musical intuition and resources available to be has been extremely gratifying. I've also grown significantly as a python data science programmer both in understanding what the language is capable of and best practices. I understand

that this is only the tip of the iceberg, and much more can be done to improve Solar Sonificiation as follows:

- Improve the commenting and organization of content in the code through peer review. Extensive comments exist in the code to explain its function, but still have not been prepared for being comprehensively shared to the public.
- Add this tutorial to Solar Protocol's library so that it can be utilized at events such as data science hackathons or in research that explores non-Western centric sonification approaches.
- Explore the possibilities of sonifications that could represent the whole dataset comprehensively while still remaining interpretable and enjoyable to listen to, rather than focusing on a few variables at a time.
    - At this time, sonifications are handcrafted per day. A common point of feedback of how to get the sounds to sound more continuous and less like discrete notes. It is possible to do so either through longer assessment of the dataset to determine what parameters are appropriate for controlling these factors or otherwise using signal processing techniques.
- Explore the possibilities of live performances and live sonifications. Live performances could work with both static data files playing through MIDI streams or live data.
    - We know it is possible to call live data from the API up to a certain time period, as new data comes in, each row of data could be sonified individually or incorporated into a continuous moving average. Additional sensors and microcontrollers could be used to generate organic human input data streams that interact with the sonification. Sonifications could also be transposed into a format designed around physical analog musical instruments rather than through digital instrumentation.

- Explore the use of additional data sources that could add more dimensions to the sonification experience, such as a weather station that could provide context for the conditions at the site or data from the position of the sun to dictate how audio should be panning spatially around the user.

8 Citations

[1] The Center for Biological Diversity, "Report: U.S. Utilities Shut Off Power 5.7 Million Times as Shareholders, Executives Raked in Billions," Press Release, Jan. 2023. Accessed: May 05, 2023. [Online]. Available: https://biologicaldiversity.org/w/news/press-releases/report-us-utilities-shut-off-power-57-million-times-as-shareholders-executives-raked-in-billions-2023-01-30/

[2] J. Lalljee, "1 in 5 adults couldn't pay an energy bill last year. It's a vicious cycle of climate crisis and companies raising prices.," *Business Insider*, Dec. 23, 2021. Accessed: May 05, 2023. [Online]. Available: https://www.businessinsider.com/cant-pay-energy-bills-higher-this-year-climate-change-responsible-2021-12

[3] C. Reinicke, "20% of Americans couldn't pay their energy bill in the last year. How to keep costs down," *CNBC*, Dec. 23, 2021. Accessed: May 05, 2023. [Online]. Available: https://www.cnbc.com/2021/12/23/20percent-of-americans-couldnt-pay-their-energy-bill-in-the-last-year.html

[4] "Solar Protocol: Exploring Energy-Centered Design," p. 7, 2022.

[5] A. Pasek, "Low-Carbon Research: Building a Greener and More Inclusive Academy," *Engaging Science, Technology, and Society*, vol. 6, pp. 34–38, Jan. 2020, doi: 10.17351/ests2020.363.

[6] "What are Affordances?," *The Interaction Design Foundation*, 2019. https://www.interaction-design.org/literature/topics/affordances (accessed Feb. 10, 2023).

[7] N. Sawe, C. Chafe, and J. Treviño, "Using Data Sonification to Overcome Science Literacy, Numeracy, and Visualization Barriers in Science Communication," *Frontiers in Communication*, vol. 5, 2020, Accessed: Jan. 12, 2023. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fcomm.2020.00046

[8] D. Greere and M. Quick, "Making numbers louder: telling data stories with sound," *DataJournalism.com*, Mar. 10, 2021. https://datajournalism.com/read/longreads/data-sonification (accessed May 05, 2023).

[9] T. Murray-Browne, "Against Interaction Design," *Tim Murray-Browne • Art ∩ Code*,

Sep. 30, 2022. https://timmb.com/against-interaction-design/ (accessed May 05, 2023).

[10] J. Silpayamanant, "Non-CWN Music Notation Software," *Mae Mai*, Aug. 24, 2013. https://silpayamanant.wordpress.com/music-notation-software/ (accessed May 05, 2023).

[11] B. K. Allami, "Microtonality and the Struggle for Fretlessness in the Digital Age," *Microtonality and the Struggle for Fretlessness in the Digital Age*, Jan. 2019. https://www.ctm-festival.de/magazine/microtonality-and-the-struggle-for-fretlessness-in-the-digital-age (accessed May 05, 2023).

[12] Cadu, *Wind Line*. 2014. Accessed: May 05, 2023. [Online]. Available: https://pioneerworks.org/exhibitions/cadu-wind-line

[13] G. Haider, "Study for a Camera on a Plot of Land in the Desert," in *Proceedings of the 2017 ACM SIGCHI Conference on Creativity and Cognition*, Singapore Singapore: ACM, Jun. 2017, pp. 432–433. doi: 10.1145/3059454.3059498.

[14] Z. Poff, *Wave Farm | Pond Station*. 2015. Accessed: May 05, 2023. [Online]. Available: https://wavefarm.org/ta/archive/works/87eejz

[15] libi rose, *autopoetic printer – libi rose (striegl)*. 2016. Accessed: May 05, 2023. [Online]. Available: https://libirose.com/portfolio/autopoetic-printer

[16] M. Parham, "Dream Lab Keynote: Marisa Parham | Price Lab for Digital Humanities," Zoom, Jun. 14, 2019. Accessed: May 05, 2023. [Online]. Available: https://pricelab.sas.upenn.edu/news/dream-lab-keynote-marisa-parham

[17] B. Braun, "Music Box Fun - Online Music Box Maker," Nov. 02, 2019. https://musicbox.fun (accessed May 05, 2023).

[18] *Sonification with Python - How to Turn Data Into Music w Matt Russo (Part 1)*, (May 02, 2022). Accessed: May 05, 2023. [Online Video]. Available: https://www.youtube.com/watch?v=DUdLRy8i9qI

[19] K. Patel, "Listen to the music of interstellar space created from NASA data - The Washington Post," *The Washington Post*, Mar. 10, 2023. Accessed: May 05, 2023. [Online]. Available: https://www.washingtonpost.com/climate-environment/2023/03/10/space-music-nasa-voyager-sonification/

[20] M. Conway, "Common Events and Function — MIDIUtil 1.1.1 documentation," Mar. 03, 2018. https://midiutil.readthedocs.io/en/1.2.1/common.html#adding-notes (accessed May 05, 2023).

[21] S. Hiehn, "How to generate music with Python: The Basics," Aug. 10, 2022.

https://scribe.rip/@stevehiehn/how-to-generate-music-with-python-the-basics-62e8ea9b99a5 (accessed May 05, 2023).

[22] A. Dingle, “Playing music from Python via fluidsynth,” 2019. https://ksvi.mff.cuni.cz/~dingle/2019/prog_1/python_music.html (accessed May 05, 2023).

9 Appendix

The current version of the code is attached. It is currently missing an example of a controller change as it was not necessary for the last sonification it produced, but does exist in previous versions of the code and will be re-incorporated in the script for tutorial purposes.

Step 0: Load Libraries

In [2]:

```
# For: generating scatter plots
import matplotlib.pylab as plt

# For: reading csvs, like our data
import csv

# For: making changes to files in the operating systems directory
import os

# For: running external programs from python
import subprocess as sp

# For: processing and interpreting the date and time of our data
from datetime import datetime

# For: accessing and sending files to a synthesizer connected to the computer
import mido

# For: working with arrays we will use to hold our data
import numpy as np

# For: converting note names to midi numbers
from audiolazy import str2midi

# For: compiling a midi file
from midiutil import MIDIFile

# For: determing what notes are in a chord
from mingus.core import chords

# alternatives: midi2str, str2freq, freq2str,freq2midi,midi2freq
# https://pypi.org/project/audiolazy/

# https://midiutil.readthedocs.io/en/1.2.1/
```

Step 1: Load Data

In [29]:

```
# Choose one date we want to sonify in YYYY-MM-DD format.
# It must be a date with an existing CSV file.
date = "2023-02-14"

# Choose the data from the folder is where all the csvs where our server data can be found relat
# In this case, I need to go out of the folder this file is in, and there I can find a folder c
# Replace it with your backup directory, the folder where your backup files are saved
backup = "../ServerBackUp/"

# Look in our backup folder for the csv file of the date selected
# Refer to it as "istr", this is an arbitrary name that refers to an file stream input
with open(backup + "tracerData" + date + ".csv", "r") as istr:
    # Parse the csv file by creating a csv object that is iterable
    reader = csv.reader(istr)

    # The first line of every tracerData backup file is the header
    # The header has a list data type which contains string objects
```

```
    header = next(reader)

    # Create an empty list for our data types
    dtype = []

    # For each column in the header, append a tuple to the array referencing that field's name
    for field in header:
        dtype.append((field, "f8"))

    # Create an empty list for the actual contents of our data too.
    # It'll be filled it with tuples, later.
    data_list = []

    # We'll need to use the date to find when midnight occurs.
    # Our data does not start or end on midnight, so we need a relative time.
    # We'll use this variable to hold that our time once we start iterating.
    midnight = None

    # So far we only read the header row
    # Let's loop through the rest of the rows and grab their data.
    for row in reader:
        # A dictionary will let us store all our rows of data and make sure each value is assoc
        row_dict = dict(zip(header, row))

        # Create an empty list for the data in the current row
        current_row = []

        # Set the datetime format used by the tracerData logs to a format string.
        # https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior
        format_str = "%Y-%m-%d %H:%M:%S.%f"

        # Python doesn't know that our datetime is a datetime yet.
        # The dictionary allows us to refer to that the first value by the column name rather t
        # Parse the string in the first value in each row, the datetime, into an interpretable
        # https://en.wikipedia.org/wiki/Unix_time
        date_obj = datetime.strptime(row_dict["datetime"], format_str)

        # With a proper time, we can now look for midnight.
        if midnight is None:
            # Midnight will now be changed to the minimum time on the date used by our datetime
            midnight = datetime.combine(date_obj.date(), datetime.min.time())
            # We'll keep the date handy as a string for our graphs
            date = date_obj.date().strftime('%Y-%m-%d')

        # To get the datetime of this row as a floating point number of seconds since 1970-01-0
        time_in_seconds = date_obj.timestamp()

        # To get the datetime a this row as a floating point number of seconds since midnight
        time_in_seconds = (date_obj - midnight).total_seconds()

        # Make this interperable version of datetime the first value we save in our current row
        current_row.append(time_in_seconds)

        # Append the rest of the values in the row as floats to the list.
        for value in row[1:]:
            current_row.append(float(value))

        # Take the list from the current_row and turn it into a tuple.
        # Add the tuple to our data_list.
        data_list.append(tuple(current_row))

        # Repeats until there are no more rows left.
```

```
# Create a structured array from our data list that will apply the data type list we created ea
# https://numpy.org/doc/stable/user/basics.rec.html
# It'll make all our data easy to reference by column names and other critera later
data_table = np.array(data_list, dtype=dtype)

# There's bug in the tracerData where time skips to an earlier time then goes back.
# Without knowing why, it seemed easiest to remove these rows with minimal impact on the data a

# Places where time skips to an earlier time will be smaller than the previous number.
# They can be found by looking for the negative differences in the datetimes.

# Find all the differences.
differences = np.diff(data_table["datetime"])

# Filter for they are less than 0.
negatives = np.where(differences < 0)

# The differences finds a value between two numbers, so it's always has one position less than
# To fix this, we'll add 1 to everything in our negative positions.
negatives = [x + 1 for x in negatives]

# This is where the negative differences actually happen relative to our original data.
print(negatives)

# Delete rows that have those positions from our data table.
data_table = np.delete(data_table, negatives)
```

```
[array([], dtype=int64)]
<class 'str'>
```

Step 2: Plot Data

These scatters will give us a high level overview of the trends of the data. The goal is not to make nice scatter plots, since our final output is audio. We can adjust the data based on what we'd like to hear, and use scatters to preview those changes before it's turned into audio.

Step 2.1: Grab Data

In [5]:

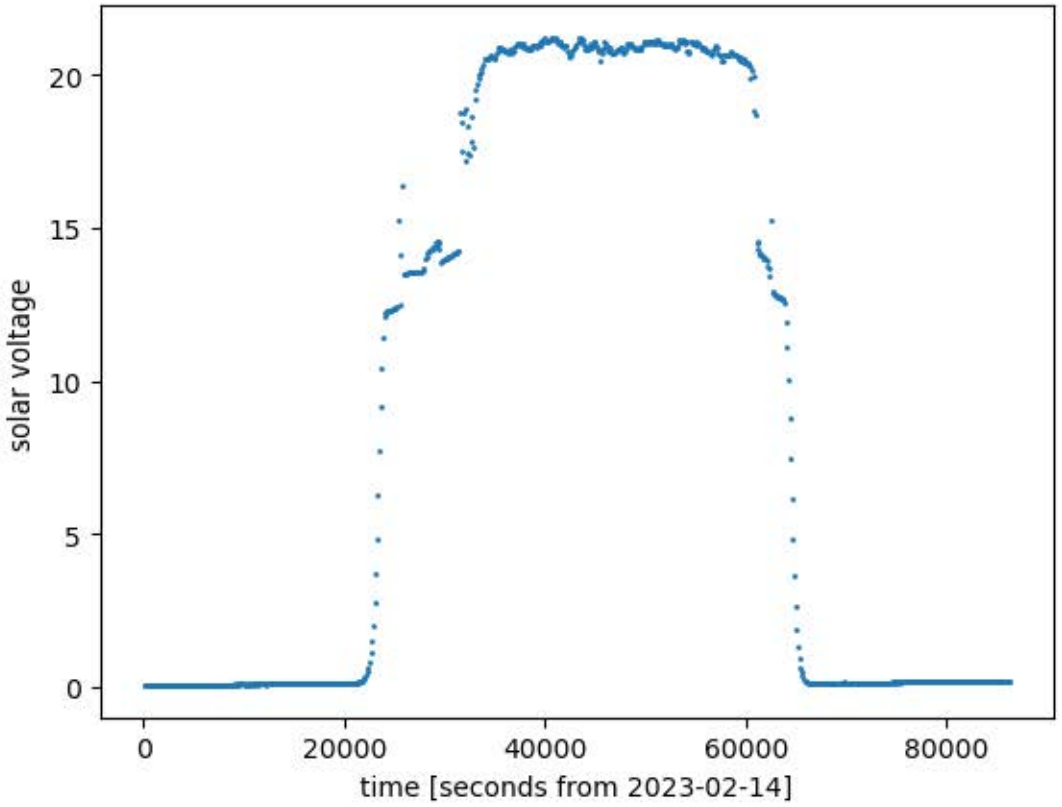
```
# Create shorter variable names for each column of data in our data table for graphing.

# time data
time = data_table["datetime"]

# panel data
solVt = data_table["PV voltage"]
solCt = data_table["PV current"]
solPw = data_table["PV power L"]

# battery data
batVt = data_table["battery voltage"]
batCt = data_table["battery current"]
batPw = data_table["battery power L"]
batPer = data_table["battery percentage"]

# load (rasberry pi) data
piVt = data_table["load voltage"]
```

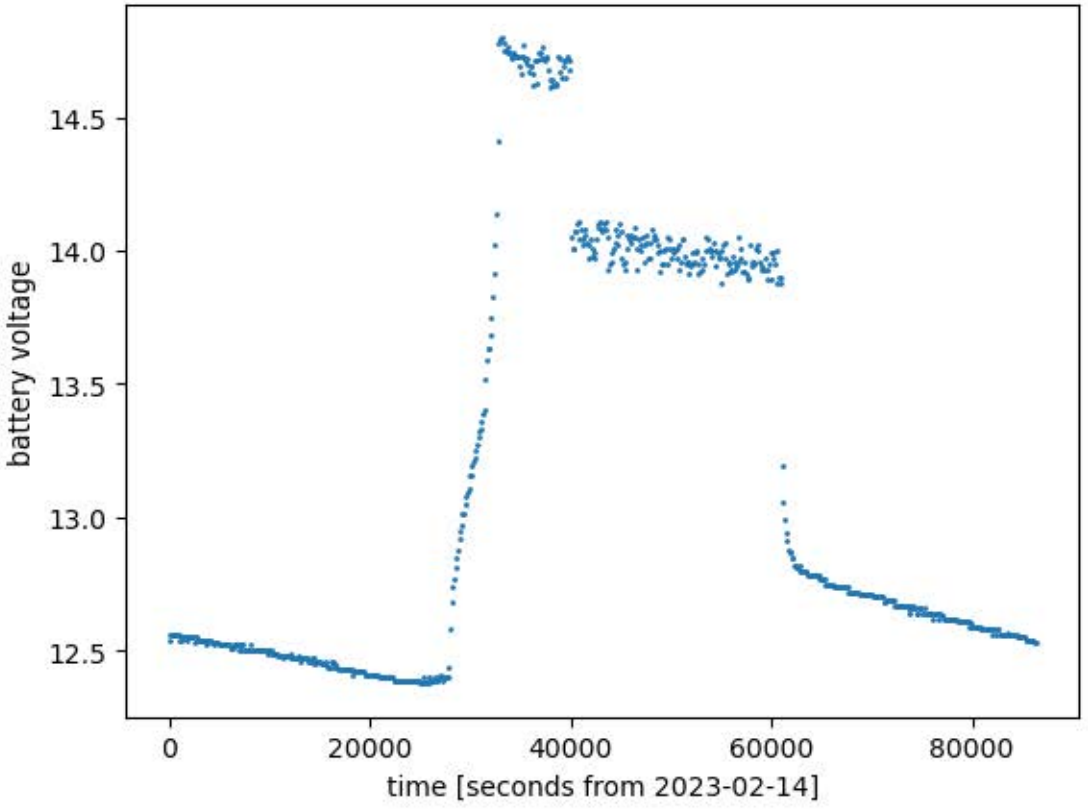
```
piCt = data_table["load current"]
piPw = data_table["load power"]
```

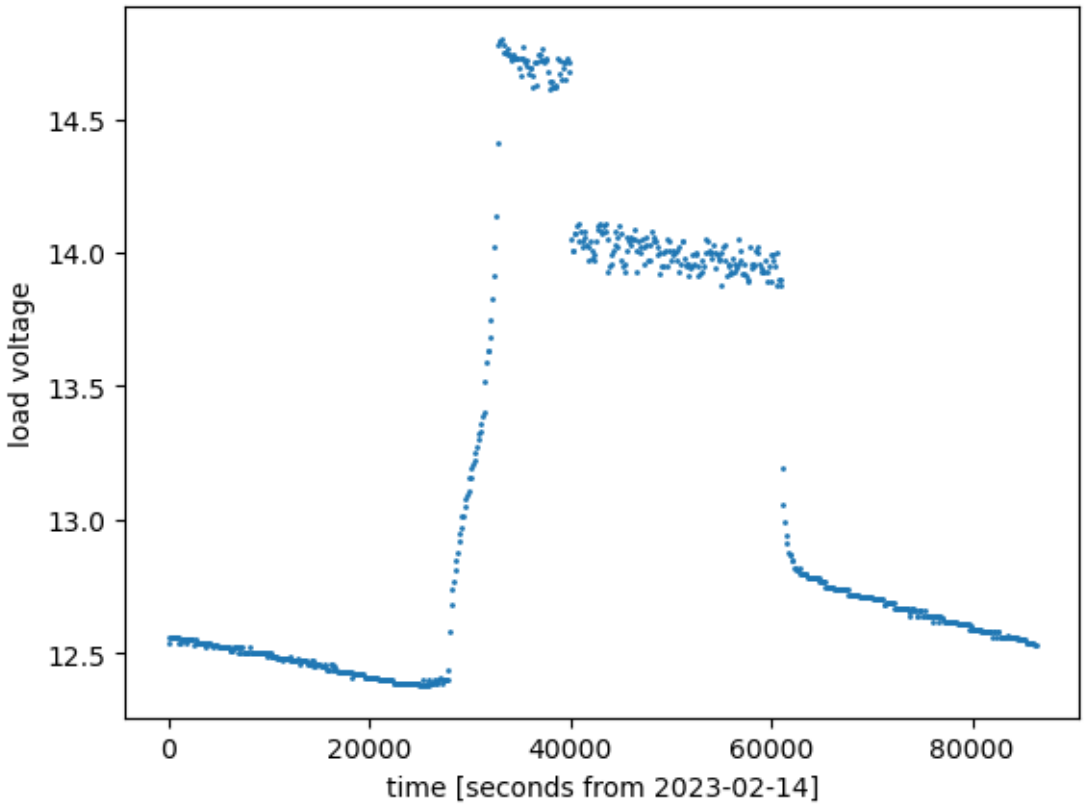
Step 2.2: Create Scatters

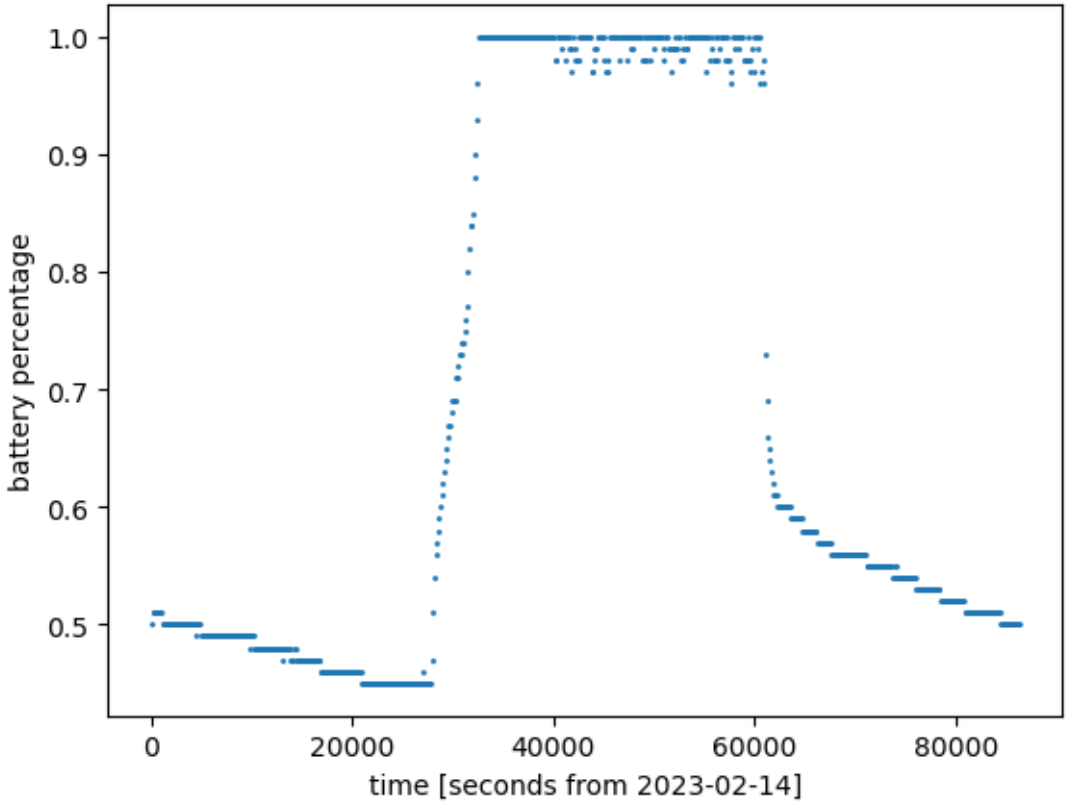
Step 2.2.1: Voltage

In [35]:
```
# Plot a scatter with our time, variable of choice, and the option to change the size of the do
plt.scatter(time, solVt, s=1)
plt.xlabel("time [seconds from "+date+"]")
plt.ylabel("solar voltage")
plt.show()

plt.scatter(time, batVt, s=1)
plt.xlabel("time [seconds from "+date+"]")
plt.ylabel("battery voltage")
plt.show()

plt.scatter(time, piVt, s=1)
plt.xlabel("time [seconds from "+date+"]")
plt.ylabel("load voltage")
plt.show()

# Percentage is not a measurement of voltage but has related trends.
plt.scatter(time, batPer, s=1)
plt.xlabel("time [seconds from "+date+"]")
plt.ylabel("battery percentage")
plt.show()


# View all voltages at once
plt.scatter(time, solVt, s=1, label="solar voltage")
plt.scatter(time, batVt, s=1, label="battery voltage")
plt.scatter(time, piVt, s=1, label="load voltage")
plt.xlabel("time [seconds from "+date+"]")
plt.ylabel("voltage [V]")
plt.legend()
plt.show()
```


solar voltage
20
15
10
5
0
0
20000
40000
60000
80000
time [seconds from 2023-02-14]


battery voltage
14.5
14.0
13.5
13.0
12.5
0
20000
40000
60000
80000
time [seconds from 2023-02-14]

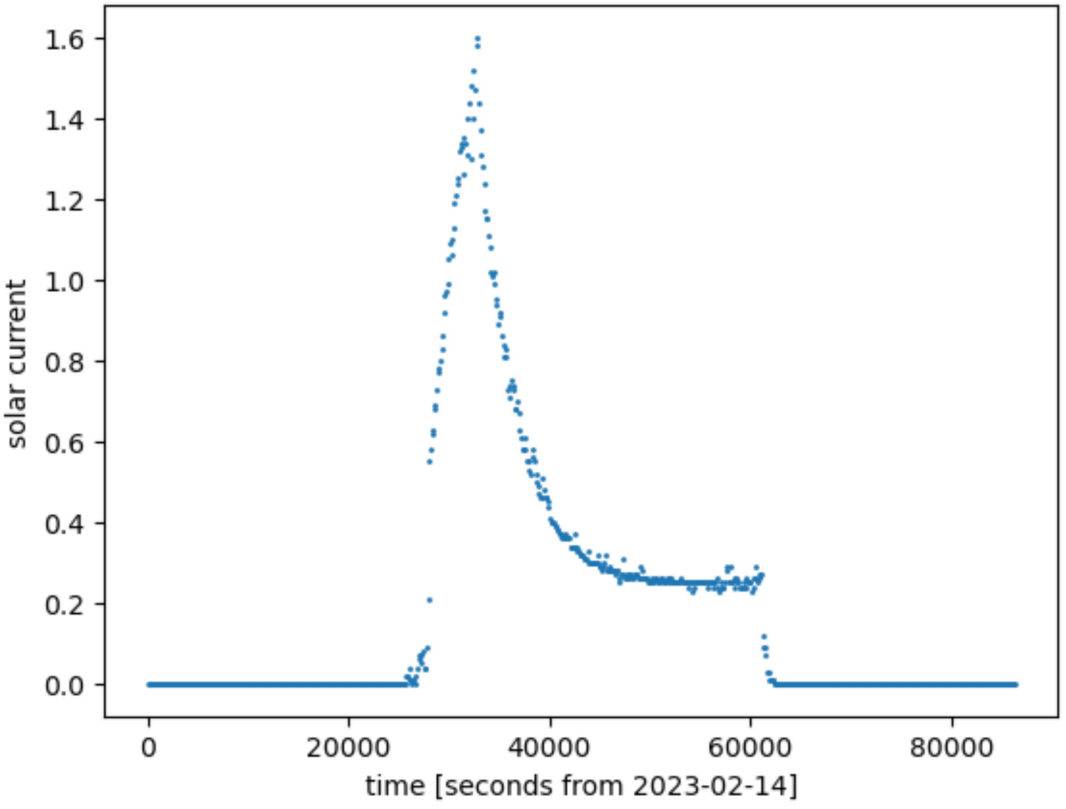


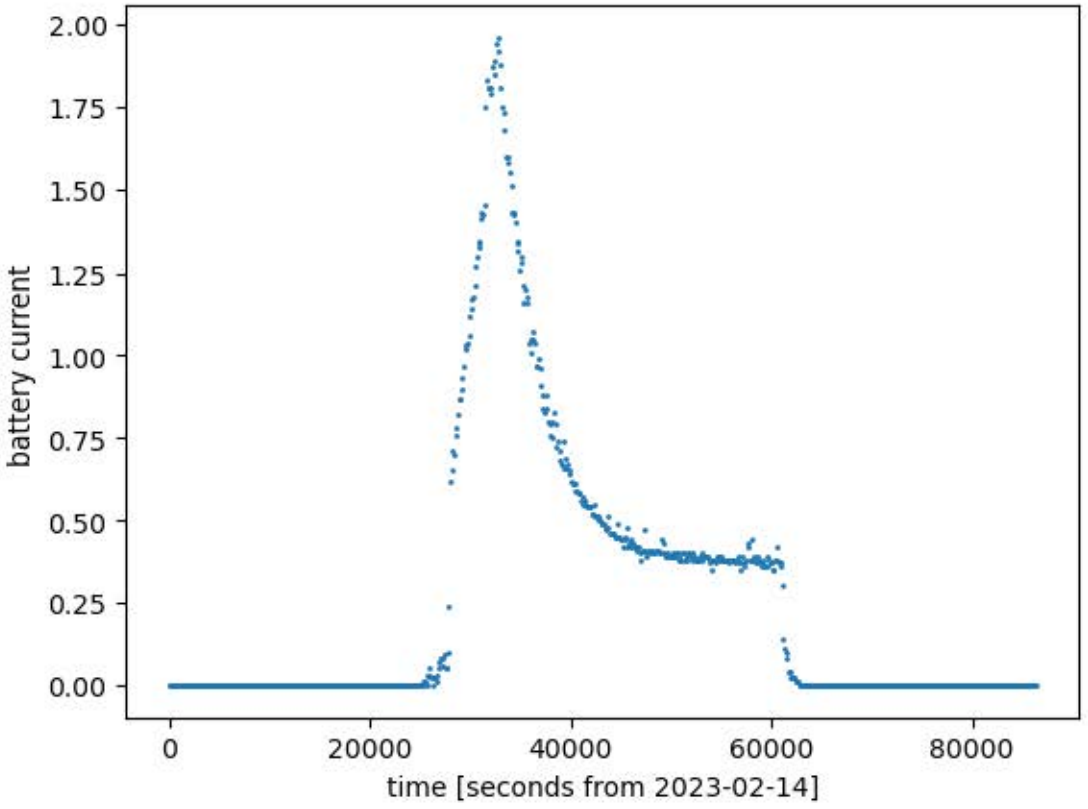


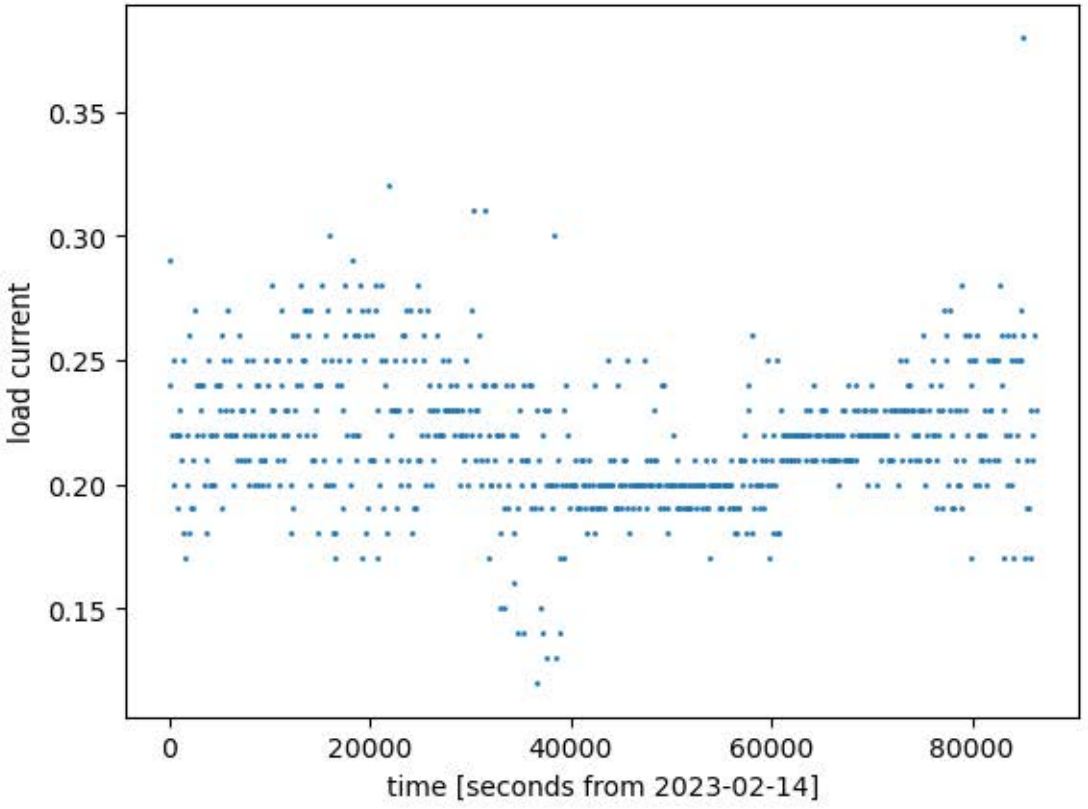
Step 2.2.2: Current

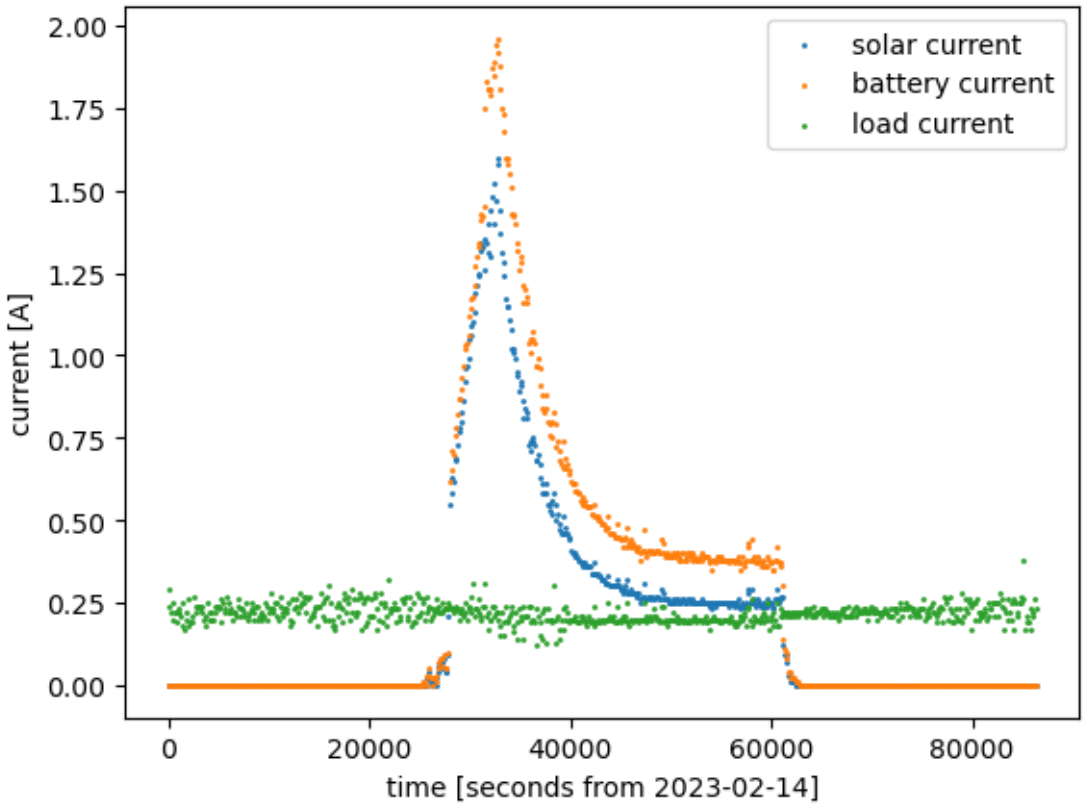
In [33]:
```
plt.scatter(time, solCt, s=1)
plt.xlabel("time [seconds from "+date+"]")
plt.ylabel("solar current")
plt.show()
```

```
plt.scatter(time, batCt, s=1)
plt.xlabel("time [seconds from "+date+"]")
plt.ylabel("battery current")
plt.show()

plt.scatter(time, piCt, s=1)
plt.xlabel("time [seconds from "+date+"]")
plt.ylabel("load current")
plt.show()

# View all currents at once
plt.scatter(time, solCt, s=1, label="solar current")
plt.scatter(time, batCt, s=1, label="battery current")
plt.scatter(time, piCt, s=1, label="load current")
plt.xlabel("time [seconds from "+date+"]")
plt.ylabel("current [A]")
plt.legend()
plt.show()
```




2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
battery current
0
20000
40000
60000
80000
time [seconds from 2023-02-14]

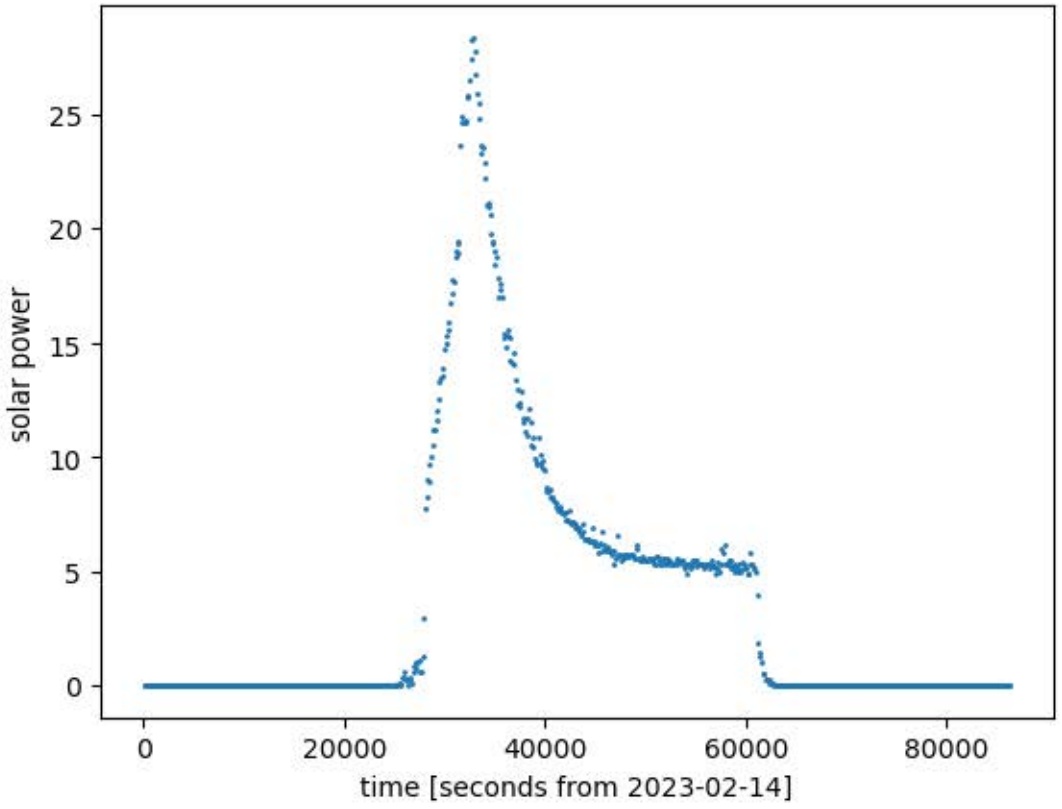

0.35
0.30
0.25
0.20
0.15
load current
0
20000
40000
60000
80000
time [seconds from 2023-02-14]

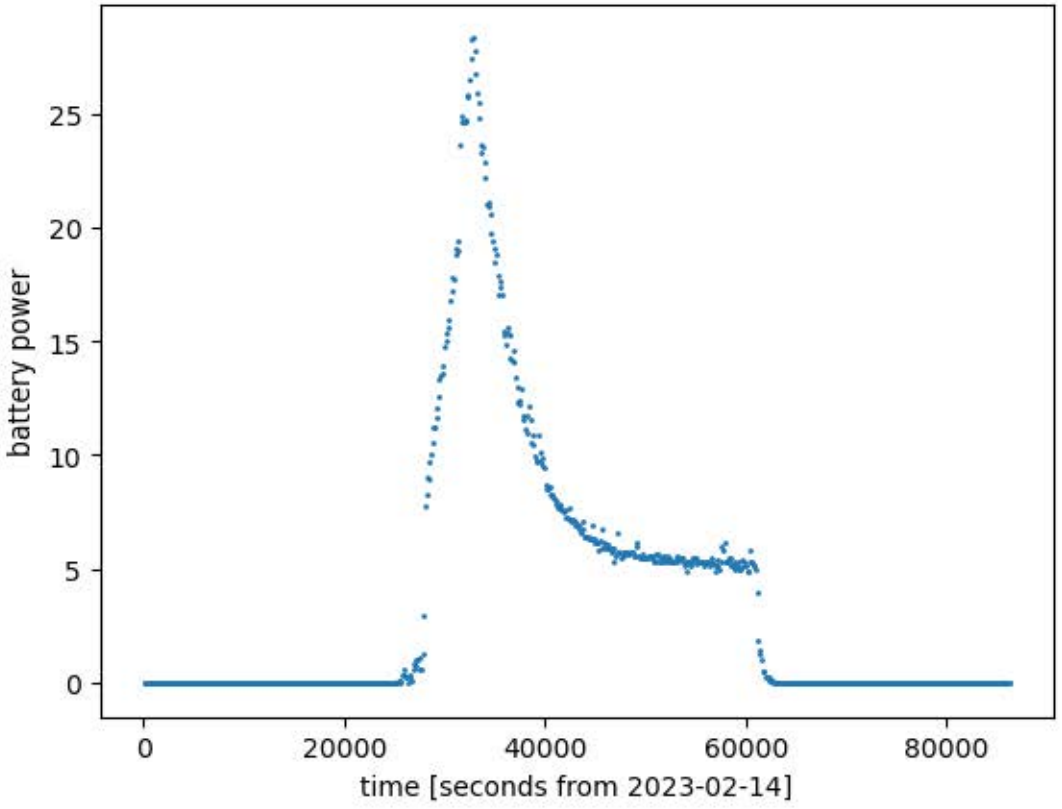


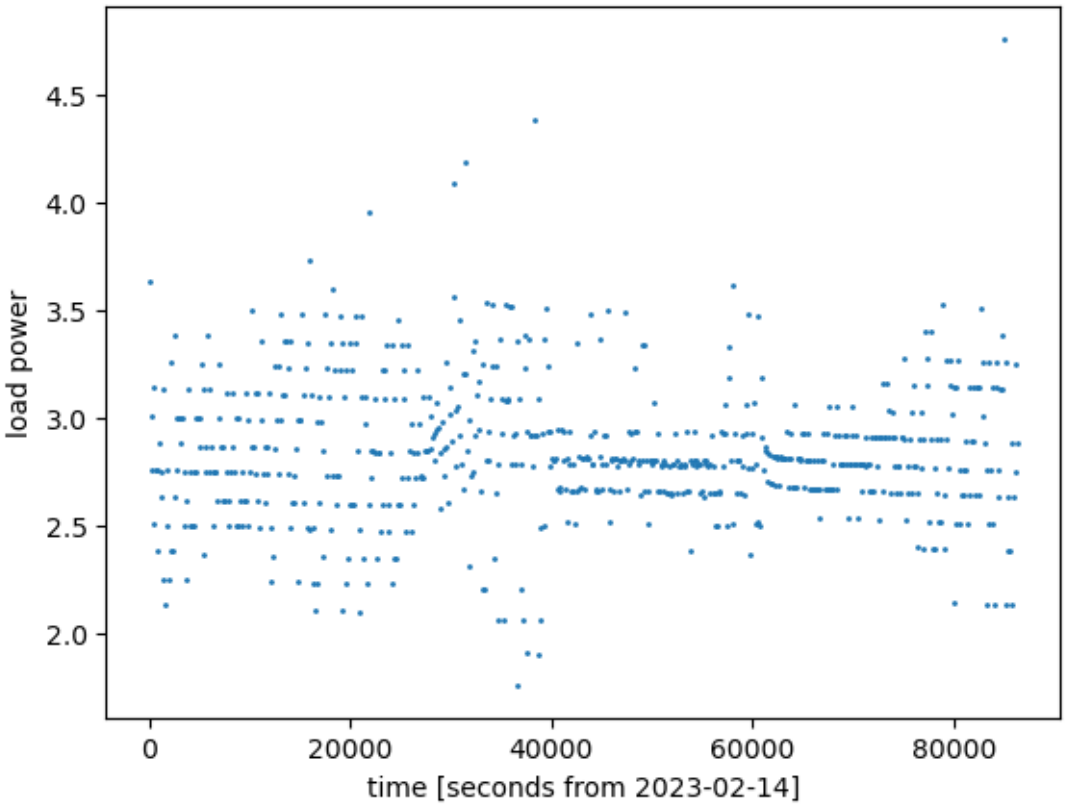
Step 2.2.3: Power

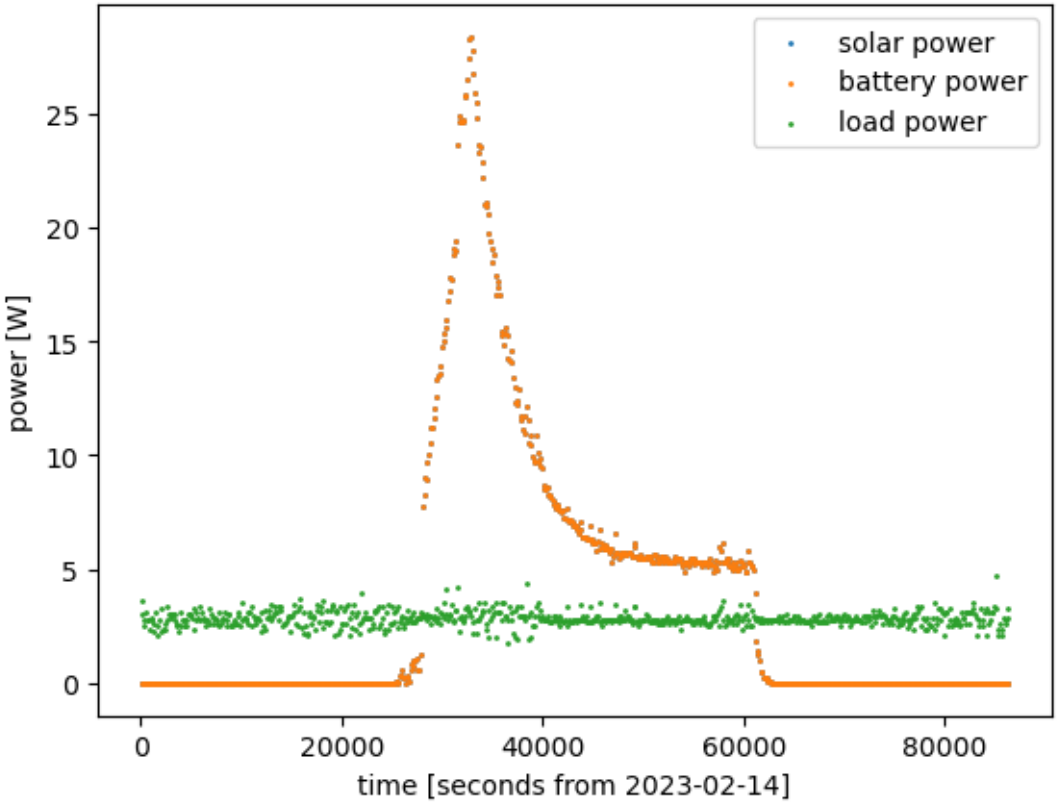
In [34]:

```
# power
plt.scatter(time, solPw, s=1)
plt.xlabel("time [seconds from "+date+"]")
plt.ylabel("solar power")
plt.show()

plt.scatter(time, batPw, s=1)
plt.xlabel("time [seconds from "+date+"]")
plt.ylabel("battery power")
plt.show()

plt.scatter(time, piPw, s=1)
plt.xlabel("time [seconds from "+date+"]")
plt.ylabel("load power")
plt.show()

# View all currents at once
plt.scatter(time, solPw, s=1, label="solar power")
plt.scatter(time, batPw, s=1, label="battery power")
plt.scatter(time, piPw, s=1, label="load power")
plt.xlabel("time [seconds from "+date+"]")
plt.ylabel("power [W]")
plt.legend()
plt.show()
```


solar power
0
5
10
15
20
25
0
20000
40000
60000
80000
time [seconds from 2023-02-14]


battery power
0
5
10
15
20
25
0
20000
40000
60000
80000
time [seconds from 2023-02-14]





Step 3: Mapping Function

The mapping function will be used to map values from one range to another. Such as from the range of our data to the range of our music parameters. For example, if we have a range of .2 to .4 for our current, 40 to 60 for our pitches, we'd want to map our original range to their closet equivalent in

the new range. For a linear approach, the minimum .2 to would become 40, max .4 would become 60, .3 in the middle would become 30.

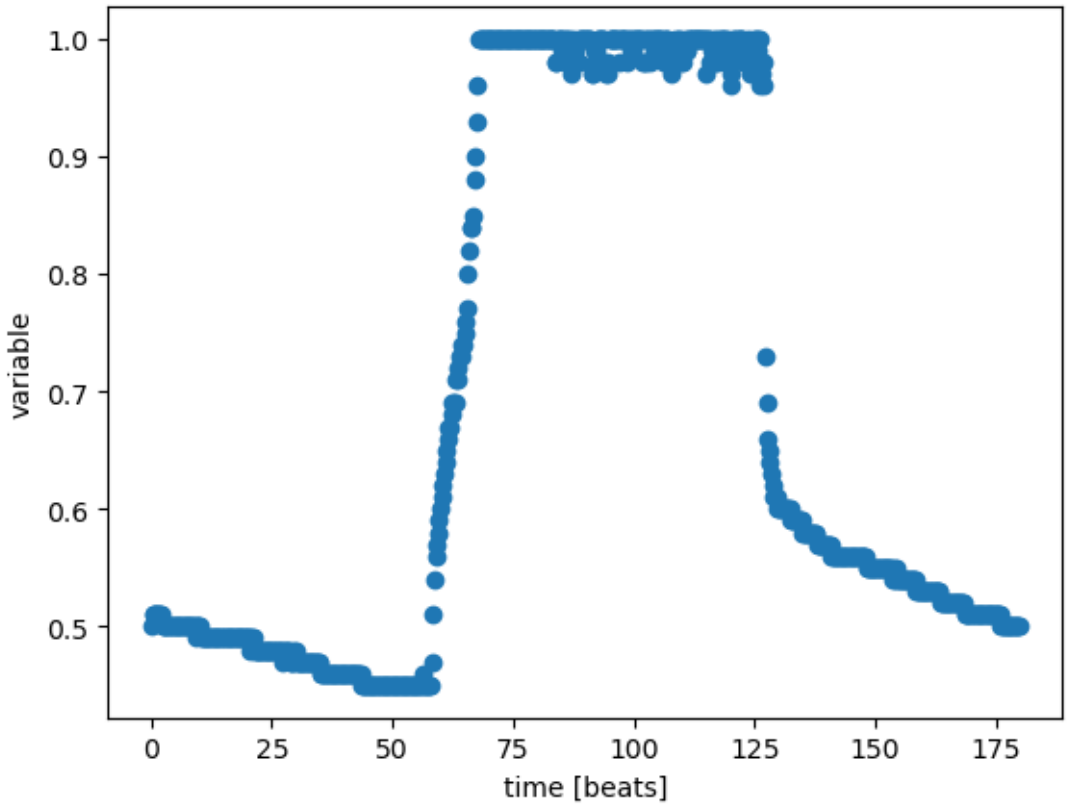
For this function: value: input value (which can be an int, float, list, or array) min_value, max_value: input value range min_result, max_result: output value range power: scaling parameter, linear mapping if set to 1. Greater than one shifts results toward the upper end of the output value range. Less than one shifts it toward the lower end of the output valume range.

In [38]:
```
def map_value(value, min_value, max_value, min_result, max_result):
    value_input = value

    # If the input value has a list data type, turn it into an array data type
    if isinstance(value_input, list):
        value = np.array(value_input)

    # Check that all the input values actually lie within the minimum or maximum range.
    # This usually isn't an issue if you use min(value) or max(value) to set the range.
    if np.any(value < min_value) or np.any(value > max_value):
        raise ValueError(
            f"one or more values is outside of range [{min_value},{max_value}]!"
        )

    # For the actual mapping, we add the mininmum output to the ratio of the input value minus
    result = (min_result + ((value - min_value) / (max_value - min_value)) * (max_result - min_

    #  If the input value was a list, turn the result from an array back to a list.
    if isinstance(value_input, list):
        result = result.tolist()

    return result
```

Step 4: Compress Time Based on Desired Time

In [41]:
```
# I arbitrarily map this to 180 seconds
# 3 minutes is more concise to listen to
# most of the action will happen during the 1-2 minute mark
# however, you could choose to do a different time scale

t_data = map_value(time, 0, 86400, 0, 180)
# beats per minute
bpm = 60
print(len(t_data))

plt.scatter(t_data, batPer)
plt.xlabel("time [beats]")
plt.ylabel("variable")
plt.show()


# If you want to add something that plays at a constant number of beats
# Create something like the following, which creates a series of numbers that have
# a constant interval for the range of your whole piece
# time_rhythm = np.arange(0, 180, 2)
```

719

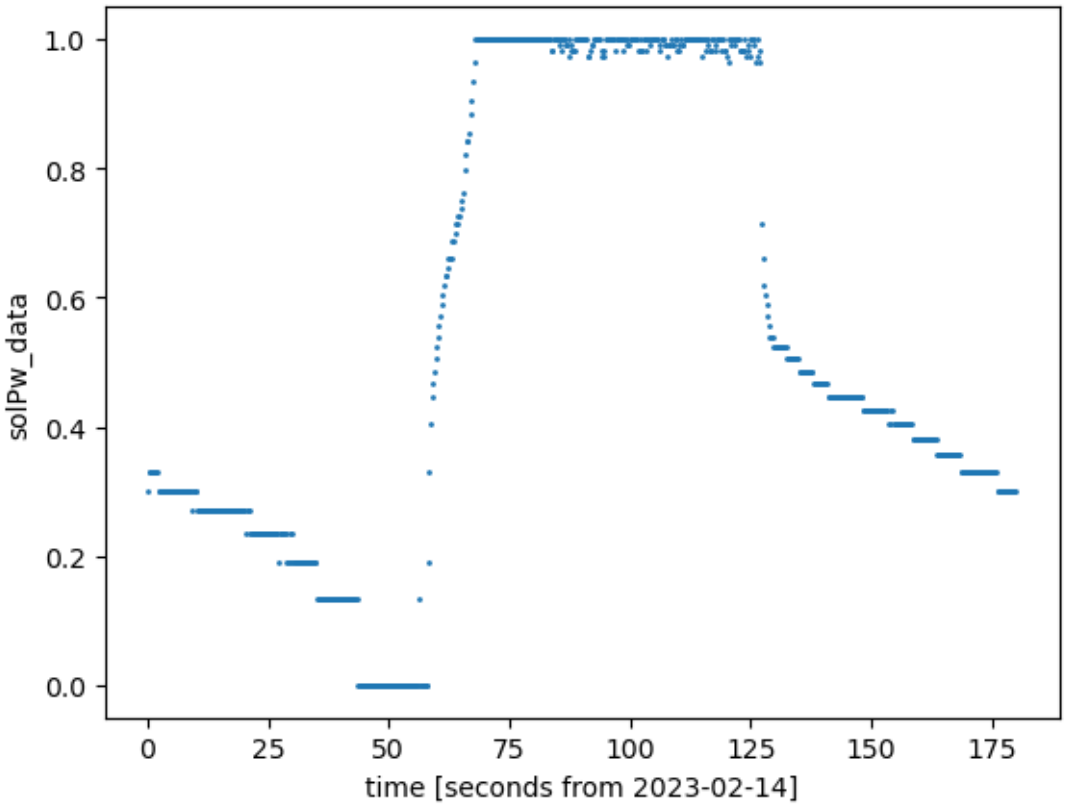


Step 5: Scale Data for Variable

Duplicate this for every additional variable you want to scale

In [42]:

```
#replace batPer with the variable you'd like to work with
#or try out the battery percentage
batPer_data = map_value(
    batPer, min(batPer), max(batPer), 0, 1
)  # normalize data, so it runs from 0 to 1

scale_1 = 0.5  # lower than 1 to spread out more evenly

batPer_data = batPer_data**scale_1

plt.scatter(t_data, batPer_data, s=1)
plt.xlabel("time [seconds from "+date+"]")
plt.ylabel("solPw_data")
plt.show()
```



Step 6: Choose musical notes for pitch mapping, convert to midi numbers

In [35]:

```
from audiolazy import midi2str, str2midi


def get_scale_notes(start_note, octaves, scale):
    """gets scale note names

    start_note: string , ex. 'C2'
    octaves: int, number of octaves
    scale: string (from available) or custom list of scale steps

    returns: list of note names (including root as highest note)
    """

    scales = {
        "chromatic": [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        "major": [2, 2, 1, 2, 2, 2, 1],
        "minor": [2, 1, 2, 2, 1, 2, 2],
        "harmonicMinor": [2, 1, 2, 2, 1, 3, 1],
        "melodicMinor": [2, 1, 2, 2, 2, 2, 1],
        "ionian": [2, 2, 1, 2, 2, 2, 1],
        "dorian": [2, 1, 2, 2, 2, 1, 2],
        "phrygian": [1, 2, 2, 2, 1, 2, 2],
        "lydian": [2, 2, 2, 1, 2, 2, 1],
        "mixolydian": [2, 2, 1, 2, 2, 1, 2],
        "aeolian": [2, 1, 2, 2, 1, 2, 2],
        "lochrian": [1, 2, 2, 1, 2, 2, 2],
        "majorPent": [2, 2, 3, 2, 3],
        "minorPent": [3, 2, 2, 3, 2],
```

```
        "wholetone": [2, 2, 2, 2, 2, 2],
        "diminished": [2, 1, 2, 1, 2, 1, 2, 1],
        # add more here!
    }

    # get scale steps
    if type(scale) is str:
        if scale not in scales.keys():
            raise ValueError(f"Scale name not recognized!")
        else:
            scale_steps = scales[scale]
    if type(scale) is list:
        scale_steps = scale

    # get note names for each scale step, in each octave
    note_names = []
    for octave in range(octaves):
        note_number = str2midi(start_note) + (12 * octave)

        for step in scale_steps:
            note_names.append(midi2str(note_number))
            note_number = note_number + step

    # add root as last note
    last_midi_note = str2midi(start_note) + (octaves * 12)
    note_names.append(midi2str(last_midi_note))

    # could alter function to return midi note numbers instead
    # note_numbers = [str2midi(n) for n in note_names]

    return note_names


note_names = get_scale_notes("C1", 3, "lydian")
note_names = get_scale_notes("C1", 3, [3, 1, 2, 2, 1, 2])  # custom scale
print(note_names)
```

```
['C1', 'D#1', 'E1', 'F#1', 'G#1', 'A1', 'C2', 'D#2', 'E2', 'F#2', 'G#2', 'A2', 'C3', 'D#3', 'E
3', 'F#3', 'G#3', 'A3', 'C4']
```

Step 7: Map data to MIDI note numbers

In [38]:

```
# choose note set
# use American pitch notation https://viva.pressbooks.pub/openmusictheory/chapter/aspn/
note_names = get_scale_notes("G4", 3, "major")
# print(note_names)

note_numbers = np.array(
    [str2midi(n) for n in note_names]
)  # make it an array so we can use do indexing on it with another array

# print(note_numbers)

# use normalized data, otherwise change 0, 1 to min max of variable
note_idx = map_value(batPer_data, 0, 1, 0, len(note_numbers) - 1)

# print(note_idx)

# rounds note index to integers
midi_data_1 = note_numbers[np.round(note_idx).astype(int)]
```

Play a sound when specific events occur

Creating chord progressions of arpeggios

https://medium.com/@stevehiehn/how-to-generate-music-with-python-the-basics-62e8ea9b99a5

In [ ]:
```
# pick a chord progression
# ealier we chose a scale and printed notes in note_names
# we can use this to choose notes for a chord progression that should sound good together becau
# use chord notation https://www.musicnotes.com/blog/a-complete-guide-to-chord-symbols-in-music/
# e.g. for G maj7
```

In [ ]:
```
NOTES = ["C", "C#", "D", "Eb", "E", "F", "F#", "G", "Ab", "A", "Bb", "B"]
# this may vary by the starting note and original range
OCTAVES = list(range(11))
NOTES_IN_OCTAVE = len(NOTES)

errors = {"notes": "Bad input, please refer this spec-\n"}


def swap_accidentals(note):
    if note == "Db":
        return "C#"
    if note == "D#":
        return "Eb"
    if note == "E#":
        return "F"
    if note == "Gb":
        return "F#"
    if note == "G#":
        return "Ab"
    if note == "A#":
        return "Bb"
    if note == "B#":
        return "C"

    return note


def note_to_number(note: str, octave: int) -> int:
    note = swap_accidentals(note)
    assert note in NOTES, errors["notes"]
    assert octave in OCTAVES, errors["notes"]

    note = NOTES.index(note)
    note += NOTES_IN_OCTAVE * octave

    assert 0 <= note <= 127, errors["notes"]

    return note
```

When system is down

If the system goes down it cannot record data. Gaps in our data where there is data missing for more than about 2-3 minutes (the approximately frequency data is recorded) implies that the system went

down. This usually occurs when the system reaches somewhere less than 34% and there's a risk of the battery discharging too much, thus the Rasberry Pi will be disconnected to conserve power. However, it could also occur for other miscellaneous reasons, like the a physical maintenance check of the system that might result in the Pi disconnected or other maintenance periods where the Pi is being updated. They may also be false gaps caused by deleting negative time differences, so you may want to print the negatives index at the start to see and want to increase the threshold of time.

This section allows you to find all the places to a MIDI event where:

- Battery goes down during the middle of the day because the battery discharged
- Battery goes down during the middle of the day because of an unknown reason
- Battery goes down during the end of the day because the battery discharged (might have false positives, because we can't check if the battery of the next day)
- Battery goes down during the end of the day because of an unknown reason

In [ ]:
```
# Similar to Step 1, we will look for differences in our data
differences = np.diff(time)

modBatDownTime = []
modUnkDownTime = []
# Filter for there are more than 3 minutes or 180 seconds gaps during the middle of the day
modDown = np.where(differences > 180)
modDown = modDown[0]
# If we have a middle of the day downtime
if len(differences[modDown]) > 0:
    # save the downtimes
    modDownTime = time[modDown]
    # check the battery percentage
    modDownTimeBatPer = batPer[modDown]
    # for every down time
    for i in range(len(modDownTime)):
        # check if the battery percentage is less than 34 and the next one is greater than .5 t
        if (modDownTimeBatPer[i] ≤ 34) & (72 ≥ batPer[modDown[i] + 1] ≥ 50):
            print(
                "There was downtime in the middle of the day at",
                modDownTime[i],
                "s UNIX time which happens at,",
                modDown[i],
                " in the data because the battery percentage was too low.",
            )
            # if yes, save this as a downtime caused by battery outage
            modBatDownTime.append(modDownTime[i])
        else:
            print(
                "There was downtime in the middle of the day at",
                modDownTime[i],
                "s UNIX time which happens at",
                modDown[i],
                " in the data for an unspecified reason.",
            )
            modUnkDownTime.append(modDownTime[i])
```

```
There was downtime in the middle of the day at 41828.144192 s UNIX time which happens at 348  i
n the data for an unspecified reason.
```

In [ ]:
```
chordProgDown = ["Gmaj7", "Amin7", "Cmaj7", "Dmaj7"]

chordProgDownNotes = []
```

```
for chord in chordProgDown:
    chordProgDownNotes.extend(chords.from_shorthand(chord))

chordProgDownNotesNumbers = []
for note in chordProgDownNotes:
    OCTAVE = 4
    chordProgDownNotesNumbers.append(note_to_number(note, OCTAVE))

# every time the system goes down because the battery ran out
# i want to play a series of notes in a chord progression
# starting from that time
chordProgDown_Time = []
print(chordProgDown_Time)
# for every time in this array
for i in range(len(modBatDownTime)):
    # I want to add enough times to play my arpeggio
    chordtimes = np.arange(
        modBatDownTime[i], modBatDownTime[i] + (len(chordProgDownNotes) * 100), 100
    )
    for i in range(len(chordtimes)):
        chordProgDown_Time.append(chordtimes[i])

chordProgDown_Time = map_value(chordProgDown_Time, 0, 86400, 0, 180)
print(chordProgDown_Time)
```

```
[]
[]
```

In [ ]:

```
eodBatDownTime = []
eodUnkDownTime = []
# Filter for where there are more than 3 minute gaps at the end of the day
# This would be the place where the last data point is at a time that's more than 3 minutes out
if (86400 - time[len(time) - 1]) > 180:
    eodDownTime = time[len(time) - 1]
    eodDownBatPer = batPer[len(time) - 1]
    if eodDownBatPer <= 0.34:
        print(
            "There was downtime in the end of the day at",
            eodDownTime,
            "s UNIX time because the battery percentage was too low.",
        )
        eodBatDownTime.append(modDownTime[i])
    else:
        print(
            "There was downtime in the end of the day at",
            eodDownTime,
            "s UNIX time for an unspecified reason.",
        )
        eodUnkDownTime.append(modDownTime[i])
```

When the system just came back online

If the system is on it can record data. If the system is coming back online from downtime caused by the battery recharging, the battery percentage should be greater than 50%, and the previous record should be less than 34%. For this code, we can't check the battery percentage that might be related to downtime that carries over from the battery running out from the night before, so we may have false positives.

This section allows you to find all the places to a MIDI event where:

- Battery comes back online during the middle of the day because the battery discharged
- Battery comes back online during middle of the day because of an unknown reason
- Battery comes back online at the start of the day because the battery discharged (might have false positives, because we can't check if the battery of the previous day)
- Battery comes back online at the start of the day because of an unknown reason

In [ ]:
```
modBatUpTime = []
modUnkUpTime = []
# if we know there's downtime during the middle of the day
if len(differences[modDown]) > 0:
    # the following index is where the downtime ends
    modUp = [x + 1 for x in modDown]
    modUpTime = time[modUp]
    # we want to see the battery percentage
    modUpTimeBatPer = batPer[modUp]
    # because we cannot iterate through floats, you may want to convert it back a percentage fl
    # if the battery percentage is greater than 50% and the previous is 34% or less
    # then we know the battery discharged and recharged
    for i in range(len(modUpTimeBatPer)):
        if (0.72 ⩾ modUpTimeBatPer[i] ⩾ 0.5) & (batPer[modUp[i] - 1] ⩽ 0.34):
            print(
                "The system came back online after downtime at",
                modUpTime[i],
                "s UNIX time which happens at,",
                modUp[i],
                " in the data because the battery percentage recovered.",
            )
            modBatUpTime.append(modUpTime[i])
        else:
            print(
                "There was downtime in the middle of the day at",
                modUpTime[i],
                "s UNIX time which happens at,",
                modUp[i],
                " for an unspecified reason.",
            )
            modUnkUpTime.append(modUpTime[i])
```

```
There was downtime in the middle of the day at 42070.531651 s UNIX time which happens at, 349
for an unspecified reason.
```

In [ ]:
```
# reverse progression
chordProgUp = chordProgDown[::-1]

chordProgUpNotes = []
for chord in chordProgUp:
    chordProgUpNotes.extend(chords.from_shorthand(chord))

chordProgUpNotesNumbers = []
for note in chordProgUpNotes:
    OCTAVE = 4
    chordProgUpNotesNumbers.append(note_to_number(note, OCTAVE))

chordProgUp_Time = []
print(chordProgUp_Time)
# for every time in this array
for i in range(len(modBatUpTime)):
    # I want to add enough times to play my arpeggio
    chordtimes = np.arange(
        # we want to make sure they play at a reasonable enough time apart so we can hear the i
```

```
        # if the time is too short the notes will just sound like a buzz
        # for every note in our chord progression, we will add a new number that is 100 seconds
        modBatUpTime[i],
        modBatUpTime[i] + (len(chordProgDownNotes) * 100),
        100,
    )
    for i in range(len(chordtimes)):
        chordProgDown_Time.append(chordtimes[i])

# we map this into the 3 minute period that we're playing
chordProgDown_Time = map_value(chordProgDown_Time, 0, 86400, 0, 180)
print(chordProgDown_Time)
```

```
[]
[]
```

In [ ]:
```
sodBatUpTime = []
sodUnkUpTime = []
# if the system is starting up after the battery has discharged the night before
# Then we want to look for significant gaps of time at the start of our data
if (time[0]) > 180:
    sodUpTime = time[0]
    sodUpTimeBatPer = batPer[0]
    if (
        0.72 ⩾ sodUpTimeBatPer ⩾ 0.50
    ):  # ideally we would like to check what the last battery value was the day before too
        print(
            "The data collection at the start of the day was delayed to",
            sodUpTime,
            "s UNIX time because the battery percentage recovered.",
        )
        sodBatUpTime.append(modUpTime[i])
    else:
        print(
            "The data collection at the start of the day was delayed to",
            sodUpTime,
            "s UNIX time for an unspecified reason.",
        )
        sodUnkUpTime.append(modUpTime[i])
```

When the battery is fully charged

The battery can be fully charged during the day. Once the battery is fully charged, the power demand tends to go down. This can be hard to see on days where there's high variability and full power isn't consistent. For now, we'll say if we see that the points before and after are greater than .95 percent, the battery is being successfully fully charged by the sun.

In [ ]:
```
# find all the times where the value of batPer is greater than .9
# find all the times where the value of batPer was previously greater than .9 but is now less t
# for times greater than .9, add a new program change, like bells
# for times less than .9, add a new program change
# if we're just doing a program change, the previous value doesn't matter
# but if we were trying to play something like a short arppegio when the system is no longer fu
# alternatively we could always skip the start of the song

pc_time = []
pc_data = []

for i in range(len(batPer)):
```

```
    if i == 0:
        pc_time.append(time[i])
        pc_data.append(1)
    if (batPer[i] >= 0.95) & (batPer[i - 1] < 0.95):
        pc_time.append(time[i])
        pc_data.append(11)
        # print(i, "full")
    if (batPer[i] < 0.95) & (batPer[i - 1] >= 0.95):
        pc_time.append(time[i])
        pc_data.append(1)
        # print(i, "no longer full")

pc_time = map_value(pc_time, 0, 86400, 0, 180)
print(pc_time)
print(pc_data)
```

```
[0.06977868333333334, 73.93394836041666, 74.4347972, 78.44145912083333, 78.69188082916666, 79.94398244166668, 80.44484795833334, 80.69527944791668, 81.02261588541666, 83.38584674166667, 85.13859505833334, 85.38902457083334, 85.6393780375, 86.14022973333334, 87.64694093958333, 88.39809051249999, 88.89877365624999, 89.64984281041667, 90.15061244791667, 90.53119489375, 98.7937599125, 99.04404499791667, 100.29602239791667, 101.29776441666665, 101.54820093125, 102.54993213958333, 102.80022421250001, 107.05707465833335, 117.82322896875, 120.07674832708334, 129.09183748125]
[1, 11, 1, 11, 1, 11, 1, 11, 1, 11, 1, 11, 1, 11, 1, 11, 1, 11, 1, 11, 1, 11, 1, 11, 1, 11, 1, 11, 1]
```

Step 9: Save data as a MIDI File

In [ ]:

```
midiF = MIDIFile(1)  # one track
midiF.addTempo(track=0, time=0, tempo=bpm)
midiF.addProgramChange(0, 0, 0, 41)

for i in range(len(t_data)):
    midiF.addNote(
        track=0, channel=0, pitch=midi_data_1[i], time=t_data[i], duration=2, volume=50
    )

filename = date + ".mid"
with open(filename, "wb") as f:
    midiF.writeFile(f)
print("saved " + date + ".mid")
```

```
saved 2023-02-21.mid
```

save midi to mp3

In [23]:

```
# equivalent to writing `fluidsynth -ni ../FluidR3_GM.sf2 2023-01-021.mid -F 2023-01-02.wav -r
filename = date + ".mid"
sp.run(
    ["fluidsynth", "-ni", "FluidR3_GM.sf2", filename, "-F", "foo.wav", "-r", "44100"],
    check=True,
)
```

Out[23]:

```
CompletedProcess(args=['fluidsynth', '-ni', 'FluidR3_GM.sf2', '2023-02-21.mid', '-F', 'foo.wav', '-r', '44100'], returncode=0)
```

In [24]:

```
mp3file = date + ".mp3"

# "-y" to overwrite, will exit if not added
```

```
# GOTO terminal: Press [q] to stop, [?] for help

# equivalent to writing `ffmpeg -i 2023-01-02.wav -vn -ar 44100 -ac 2 -b:a 192k output2.mp3 in
sp.run(
    [
        "ffmpeg",
        "-y",
        "-i",
        "foo.wav",
        "-vn",
        "-ar",
        "44100",
        "-ac",
        "2",
        "-b:a",
        "192k",
        mp3file,
    ],
    check=True,
)

os.unlink("foo.wav")
```

Optional: Play the MIDI stream through a synthesizer

In [25]:
```
# use this to see what ports are available
mido.get_output_names()
```

Out[25]:
```
['Microsoft GS Wavetable Synth 0']
```

In [ ]:
```
# play the midi file using the synthesizer of your choice, it will close the port after it's do
with mido.open_output("Arturia MicroFreak 1") as port:
    mid = mido.MidiFile(filename)

    for msg in mid.play():
        port.send(msg)
```

Optional: Listen to MIDI file within jupyter

In [ ]:
```
# For: playing MIDI files
# https://pypi.org/project/pygame/
import pygame

# start up
pygame.init()

# choose song
pygame.mixer.music.load(filename)

# play song
pygame.mixer.music.play()
```

In [ ]:

```
# stop playing song
pygame.mixer.music.stop()  # Comment this if you want to keep listening.
```