Asynchronous Input Buffers Control Regime

Sorry for the delay in getting this to you. In the process of generating recording data I became aware of the LibAiff library not properly writing samples to AIFF file. Besides the effort required to find and implement a replacement, this discovery was not devastating. LibAiff was too cumbersome of an API and had insufficient documentation anyway. As a replacement, the program now uses libsndfile[1], which is much more mature in documentation and functionality, and most importantly, records about as well as the CSV file stream.

Per your request, for each implementation a diagram is provided for description of the process flow used for recording using asynchronous input/output (AIO). Refer to the header file `recorder_helpers.h` and its corresponding '.c' `recorder_helpers.c` file for the custom wrapper function names used below. If the functions referenced are not listed there, refer to the manufacturer's header file `dt78xx_aio.h` and its corresponding '.c' file `dt78xx_aio.c`. All relevant code can be found in the `INSTALL/recorder/` directory of the repository. Lastly, a PNG version of each diagram is available in the directory this document is located in the `diagrams/` directory (the names are `original.png` and `current.png`) of the `.zip` file containing this document.

Explicit Double Buffer: Ping-Pong

The so-called 'original' explicit implementation of a double buffer, or ping-pong buffer, is described graphically below in Figure 1. The leftmost third of the flow chart relates to the setup and configuration of the AIO stream, channels, triggers, and buffers. It also concerns the feature for cycling on and off according to sunset and sunrise times, and desired duration of recording. As such, I won't go into detail on these features as they are not relevant. The remaining two-thirds is the ping-pong buffer mechanism.

Once the recording (on) cycle is enabled, the program follows the algorithm outlined below. To aid in mapping them to Figure 1 the specific function is included in parentheses. The steps are:

1. Create a file descriptor for the AIFF to be written to (`createAIFF()`, which calls libsndfile's public function).
- **1.1.** If just entering the recording `while` loop, that is, it just became 'night', then:
  - **1.1.1.** Remember it is night (`night = 1`).
  - **1.1.2.** Arm and start the input stream (`armStartStream()`).
  - **1.1.3.** Remember it is the first buffer to be written to file.

[1] https://en.wikipedia.org/wiki/Libsndfile

1.2. Otherwise, go to **1.1.3** and continue to **2**.

2. Wait indefinitely (`aio_wait(-1)`) until one of the ping buffer has been completely filled by the input stream.

2.1. If the buffer is not complete (`aio_wait() < 1`), go back to **2**.

2.2. Otherwise, proceed to **3**.

3. Increment the number of file buffers completed.

3.1. If the number of buffers completed is equal to the number of buffers to be written per file, then go to **4**.

3.2. Otherwise,

3.2.1. If the number of buffers is even, then write the ping buffer (`write(PING)`) whilst indefinitely waiting for the pong buffer to fill completely (`aio_wait(-1)`). When full return to **2**.

3.2.2. Otherwise (the number of buffers is odd), write the pong buffer (`write(PONG)`) whilst continuing and returning to **2**.

4. Clean up after writing to file by closing the file descriptor (`finishAIFF()`), and then return to **1** if still night. Otherwise wait until night and proceed with **1**.

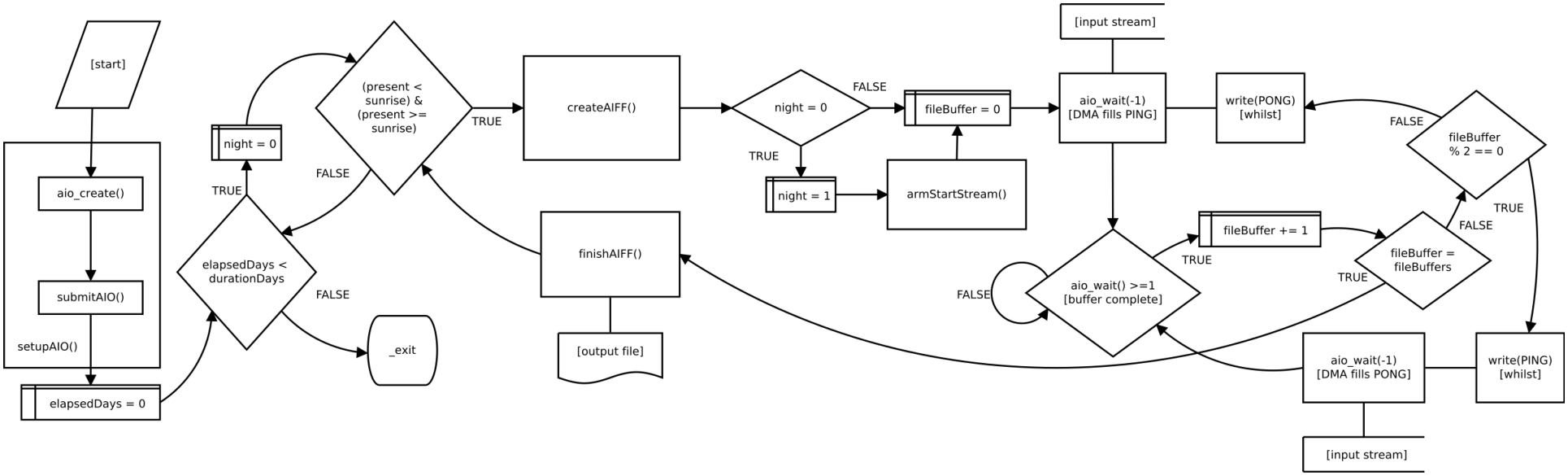


Figure 1: Process flow for the ping-pong buffered implementation. Rhombi correspond to conditional if-else logic and rectangles correspond to helper and API function calls. Text in brackets are non-code annotations included for readability. An arrow indicates direction of process flow and a line indicates a joint process that occurs at the same time.

The control buffers of the AIO (ping and pong) are requeued for filling after being read and written from, in the act of setting up the input stream, the stream is set to requeue buffers (`aio_create(inStream, 0, isInAIODone, isInStreamEmpty)`, where the call back `isInAIODone`. This function is called when a buffer is complete, and always returns 1 after writing to file (do some 'work'), which enables requeuing for input stream `inStream`). This step is essential for the ping-pong buffering to work as designed.

Implicit Double Buffer: AIO

Before launching into describing this improved process flow, first a brief comment. This implementation has the added flexibility of increasing the number of 'queued' buffers to quantity larger than 2 (at most 128). It is advisable to increase the number of queued AIO buffers (at least two) as you use higher sampling frequencies. I found 16 works well. The program automatically sizes the buffers as to not exceed the 16-bit sampling limit, which we discovered earlier.

The so-called 'updated' or 'current' implicit implementation of a multi-buffer regime is described graphically below. Like in the previous implementation, the leftmost third of the diagram predominantly sets up and configures the input stream. Unlike in Figure 1, the leftmost third of Figure 2 has been expanded to show the process in the `setupAIO()` that enables requeuing of buffers after they have been read and written to file. This new implementation is notably more complex, but more compact, owing to the fact that the buffering is implicitly performed by the API provided by the manufacturer, rather than explicitly being handled in `main`.

The algorithmic steps (baring the setup and time dependent conditionals) are as follows:

1. If just entering night (`night = 0`) proceed to **2**, else, skip to **3**.
2. Arm and start the input stream (`armStartStream()`). Also remember it is night now (`night = 1`).
3. Asynchronously sample input stream, blocking for 0 milliseconds (`aio_wait(0)`) and return 1, whilst simultaneously the control buffers submitted (`submitAIO()`) allocated in setup (`setupAIO()`) are filled. That is:
    3.1. The first queued buffer is filled with input stream data.
    3.2. Once done, or entirely full, queue the next buffer to be completed, returning to **3.1** and proceeding, and at the same time continuing to the next step.
    3.3. As you continue to write the samples in the previous buffer to file, with this new, current buffer go to **1**.
4. If this is the first buffer to be written to file, then:
    4.1 Create an AIFF file (`createAIFF()`) and continue to the next step.

4.2 If otherwise, skip to next step.

5. With the data in this buffer, write it to the AIFF file created in **4.1**.

6. Increment by one and remember the number of buffers written for this file.

6.1. If the number of buffers written is equal to the number to be written, then skip step **7** and proceed.

6.2. Clean up after the AIFF file by closing its file descriptor (`finishAIFF()`).

6.3. Set and remember the number of buffers written to the now closed file as zero.

7. Requeue the now completed buffer so it can be used again to store input samples.

8. Repeat by going to **3** and continue yet again.

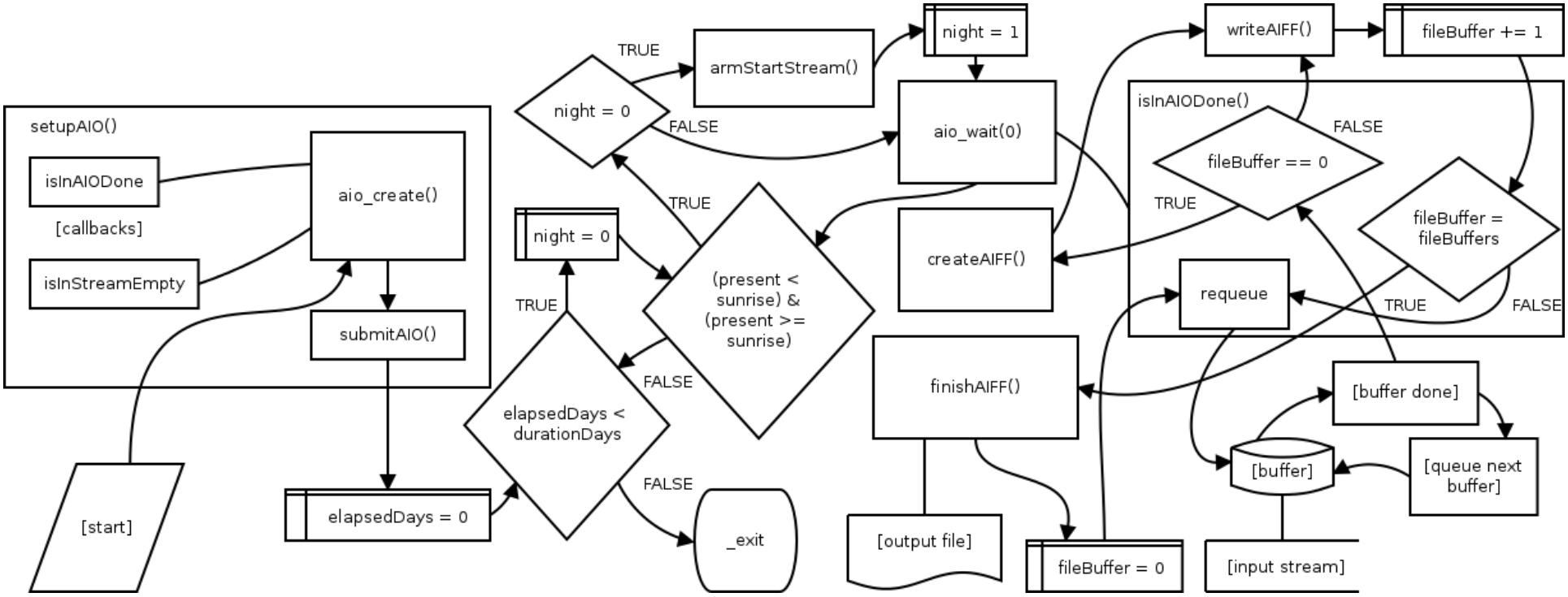


Figure 2: Process flow for the AIO multi-buffered implementation. Rhombi correspond to conditional if-else logic and rectangles correspond to helper and API function calls. Text in brackets are non-code annotations included for readability. The encompassing rectangles indicate a broader in scope helper function that calls other functions within its scope. An arrow indicates direction of process flow and a line without an arrow head indicates a joint process that occurs at the same time.

This process allows data to be stored simultaneously as data is written to file *ad infinitum*, or until night ends, etc. The asynchronous input capability is permitted by Direct Memory Access (DMA) and its multi-threading. The key difference between this implementation with the original one is that the API that interfaces with the input control buffers is used to not only requeue the buffers but also write to file using the `isAIODone()` function, which is called back at control buffer completions. Refer to `dt78xx.h` and `dt78xx.c` for explanation on the AIO API, in addition to `aio-out` in the `example-applications/dt78xx-examples/aio-out/` directory, which has helpful comments.

Sampling Frequency Sets

For both CSV and AIFF file recording demonstrations, plots of the recorded data in the temporal and spectral domain are provided. Each recording is 60 seconds long. All of the recordings were made using the implicit double buffer AIO. Under either file formats are the sampling frequency and two demonstrative analog signals. Each signal, which is defined by its frequency, amplitude, and waveform, has a file corresponding to its recording is provided in addition to the time- and frequency-domain plots. For future reference, the best way for quick visual and auditory inspection and confirmation is opening AIFF files Audacity.

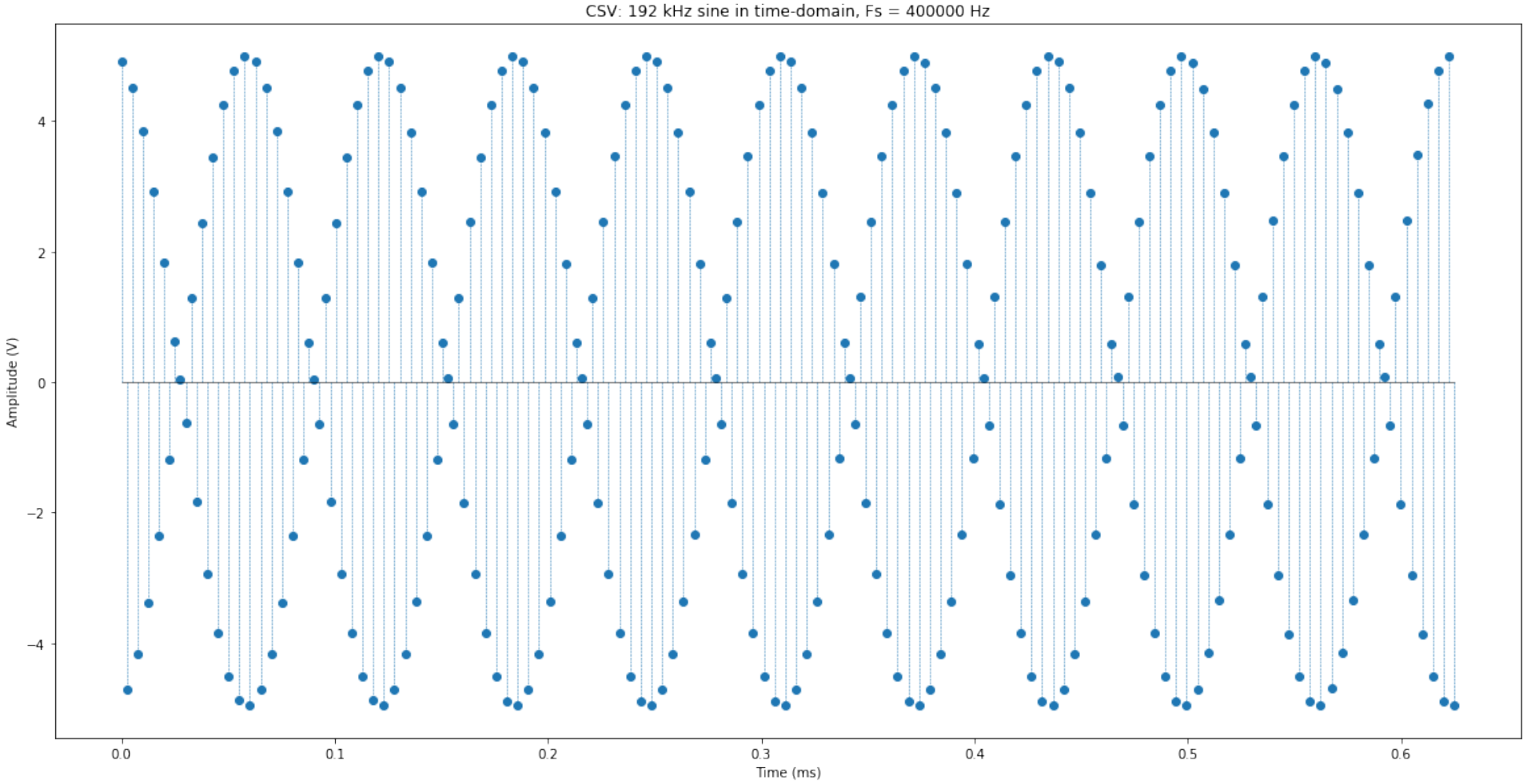
Before showing the plots, a few concerning observations. Both the CSV and AIFF recordings suffer from strange ‘beat’ patterns when sampling at close to the Nyquist frequency. These patterns also occur under the same conditions using the `aio-in` example program. However, the frequency content of the beat waveform is a pure sinusoid with the correct frequency so I am unsure what is the going on. Interestingly, if the sine waveform frequency is made a quarter that of the sampling frequency instead of less than but about half, the beat pattern disappears and the recording is a pure sign wave as should be expected. Finally, libsndfile's AIFF writer results in recordings similar but inferior to the CSV ones, although at considerable slower write speed that has introduced drop-offs into the recording after moving to the next queued buffer. (These drop-offs are best seen in Audacity rather than the plots.)

All the plots were made with Python’s Matplotlib. You can change the range of samples shown in the Jupyter notebook `plot_csv_aiff.ipynb` in the `.zip` folder. The plots below do not show all or near all of the samples as there are just too many samples in a 1-minute to show on one plot. I had made a MATLAB script that could display sequences of frames containing only the CSV recordings since Audacity cannot load the file. These have also been included (`plot_csv.m` and `plot_aiff.m`), and have only been run in GNU Octave (the open source MATLAB clone)—I need to translate them to Python, sorry. Alas, I am disappointed with the poor, albeit close, results—the periodic drop offs—from the AIFF recordings since they are absent in the CSV recordings.

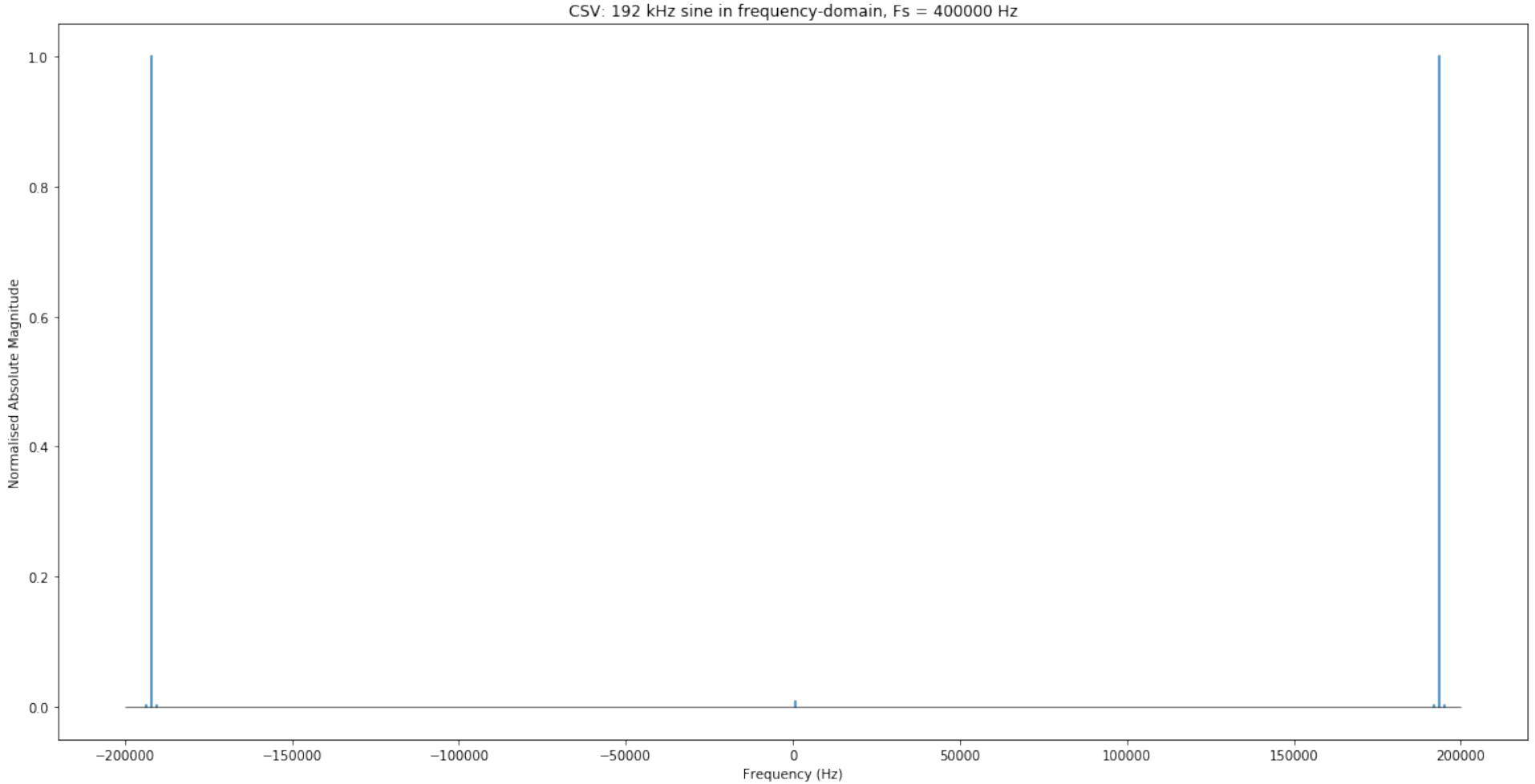
csv

400 kHz

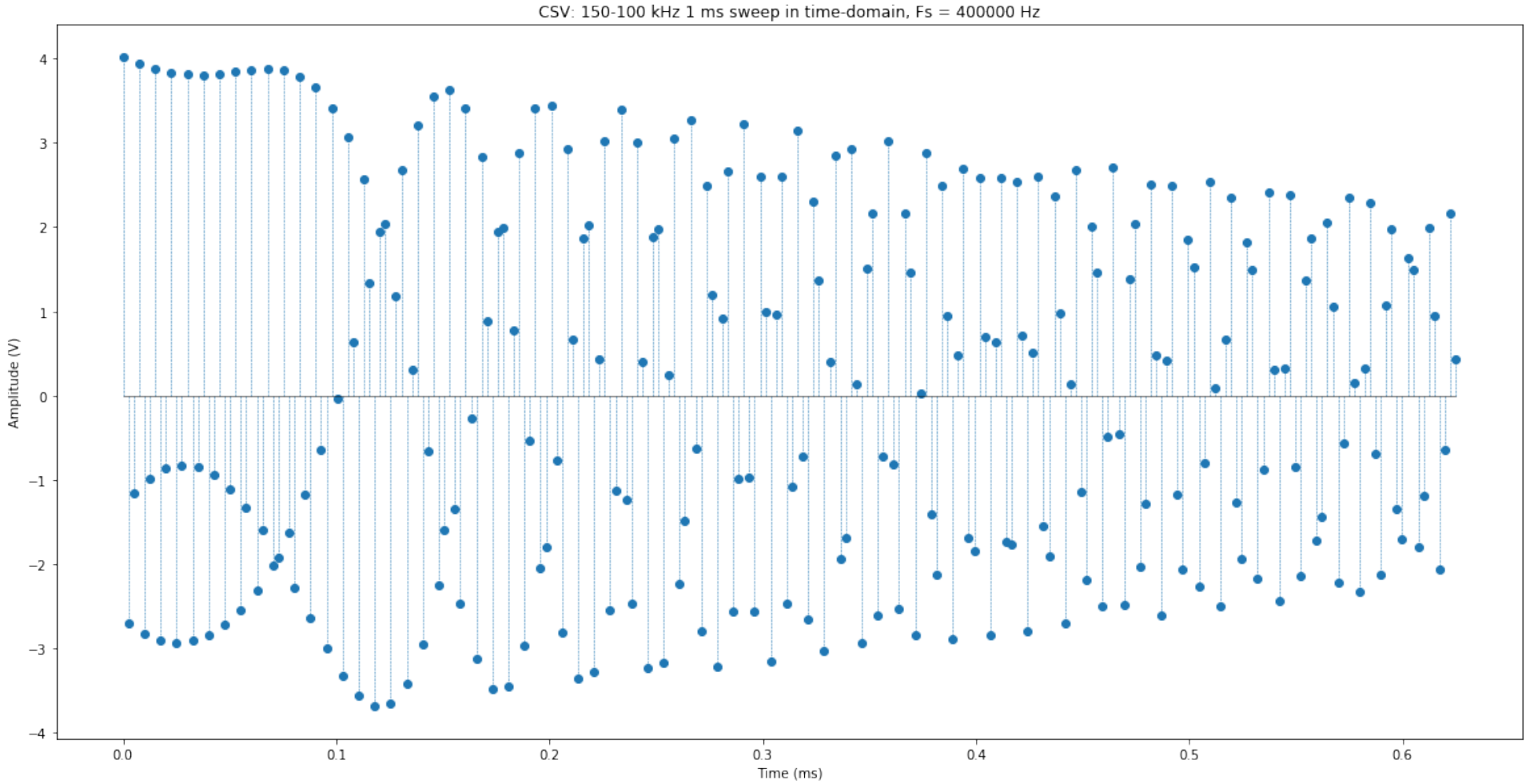
1) 192 kHz 5 V Sine
   - Recording file: csv/400/sine/192kHz400_20131219T021907434105Z.csv
   - Temporal domain recording



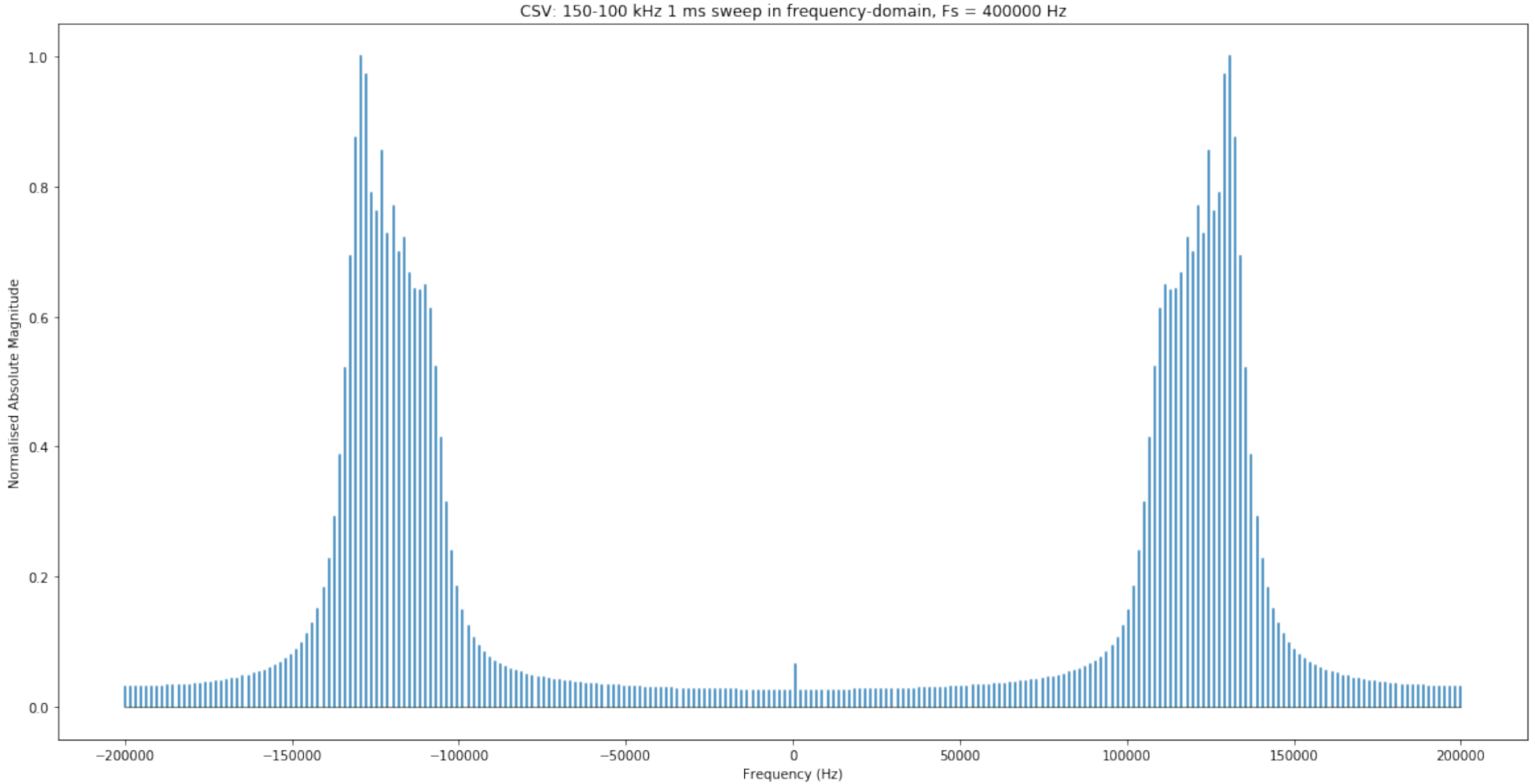
- Spectral domain recording:



2) 150 kHz 5 V to 100 kHz 2 V Sine 1 ms Sweep
   - Recording file: csv/400/sweep/150-100kHz400_20131219T021421308345Z.csv
   - Temporal domain recording:

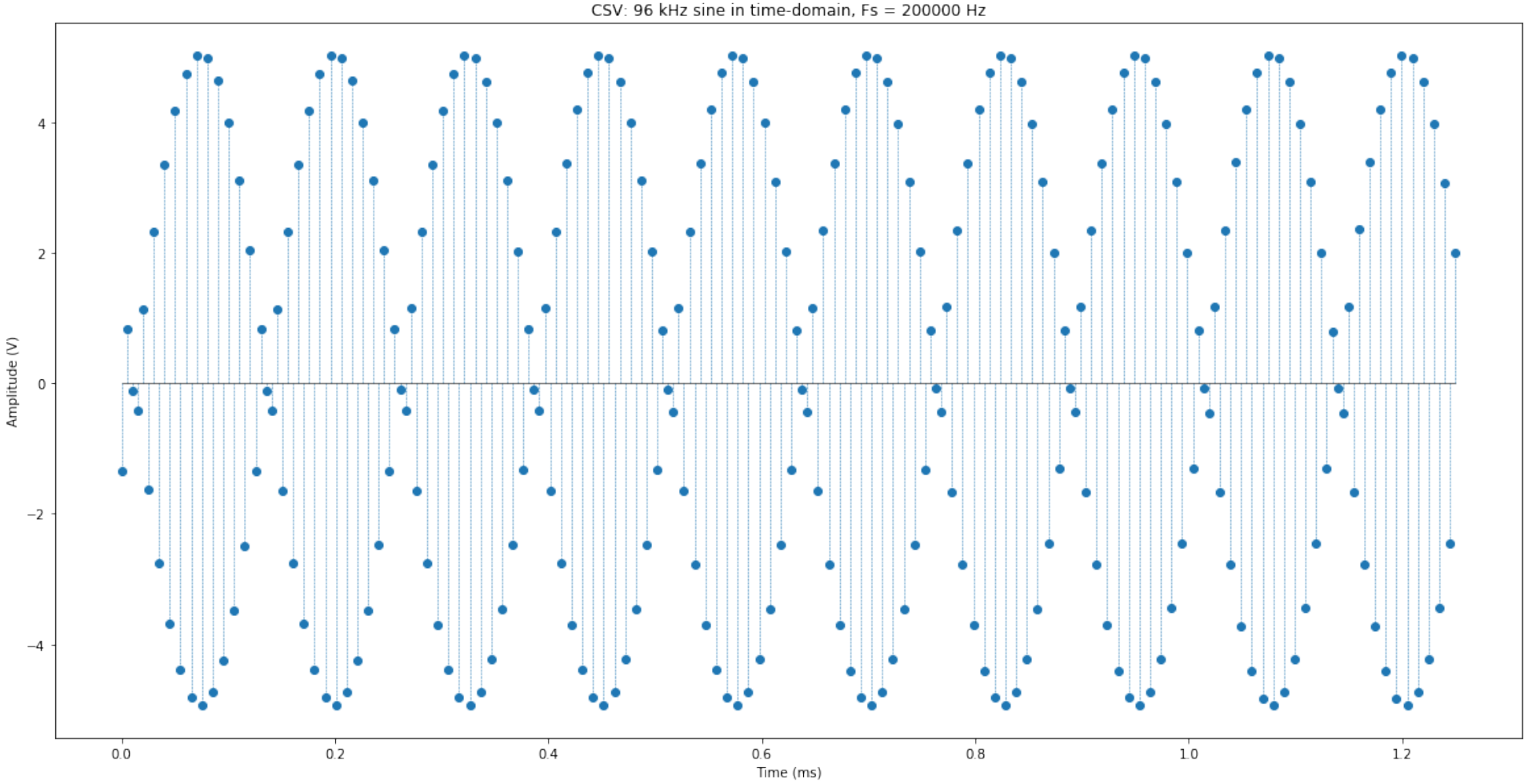


- Spectral domain recording:

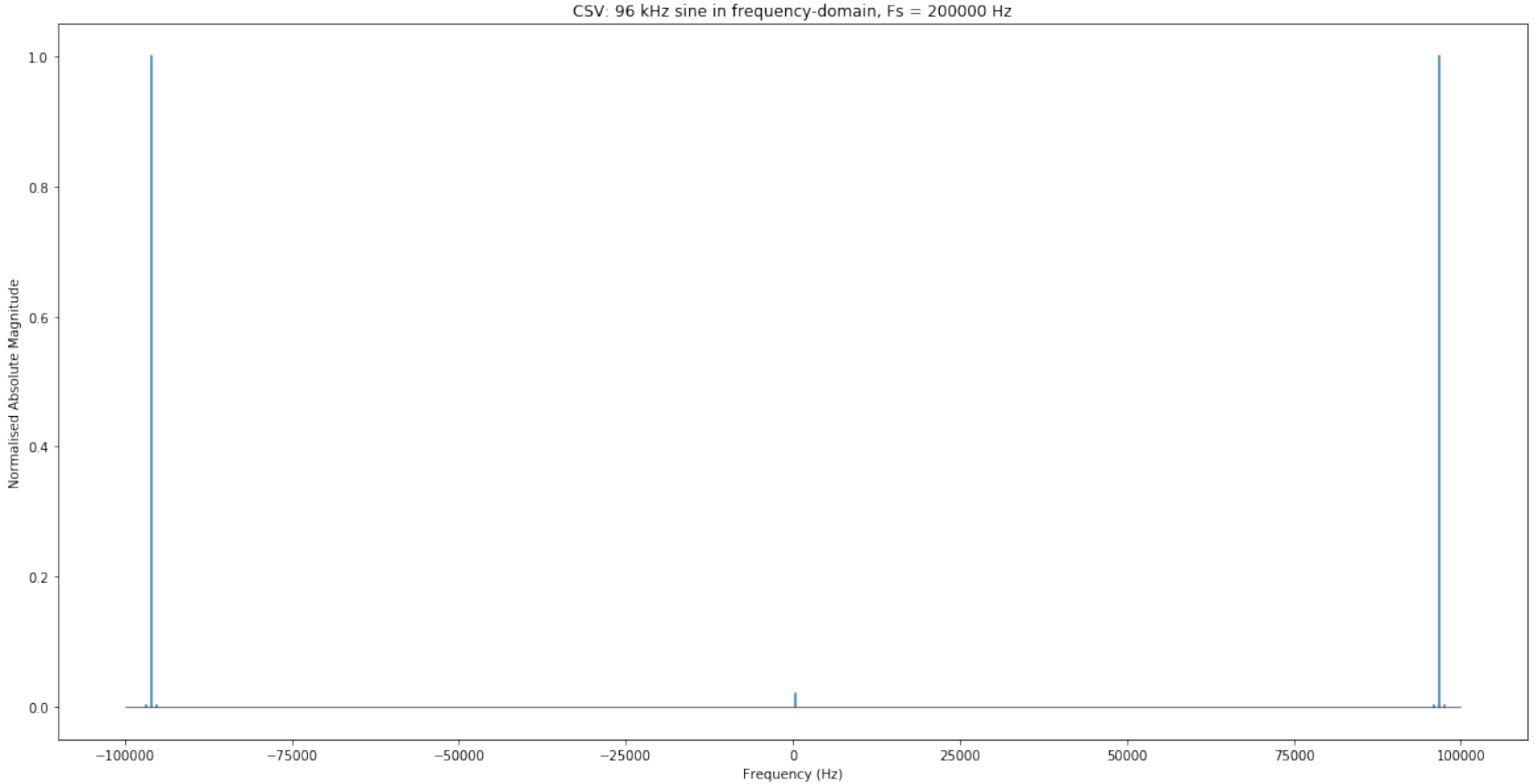


200 kHz

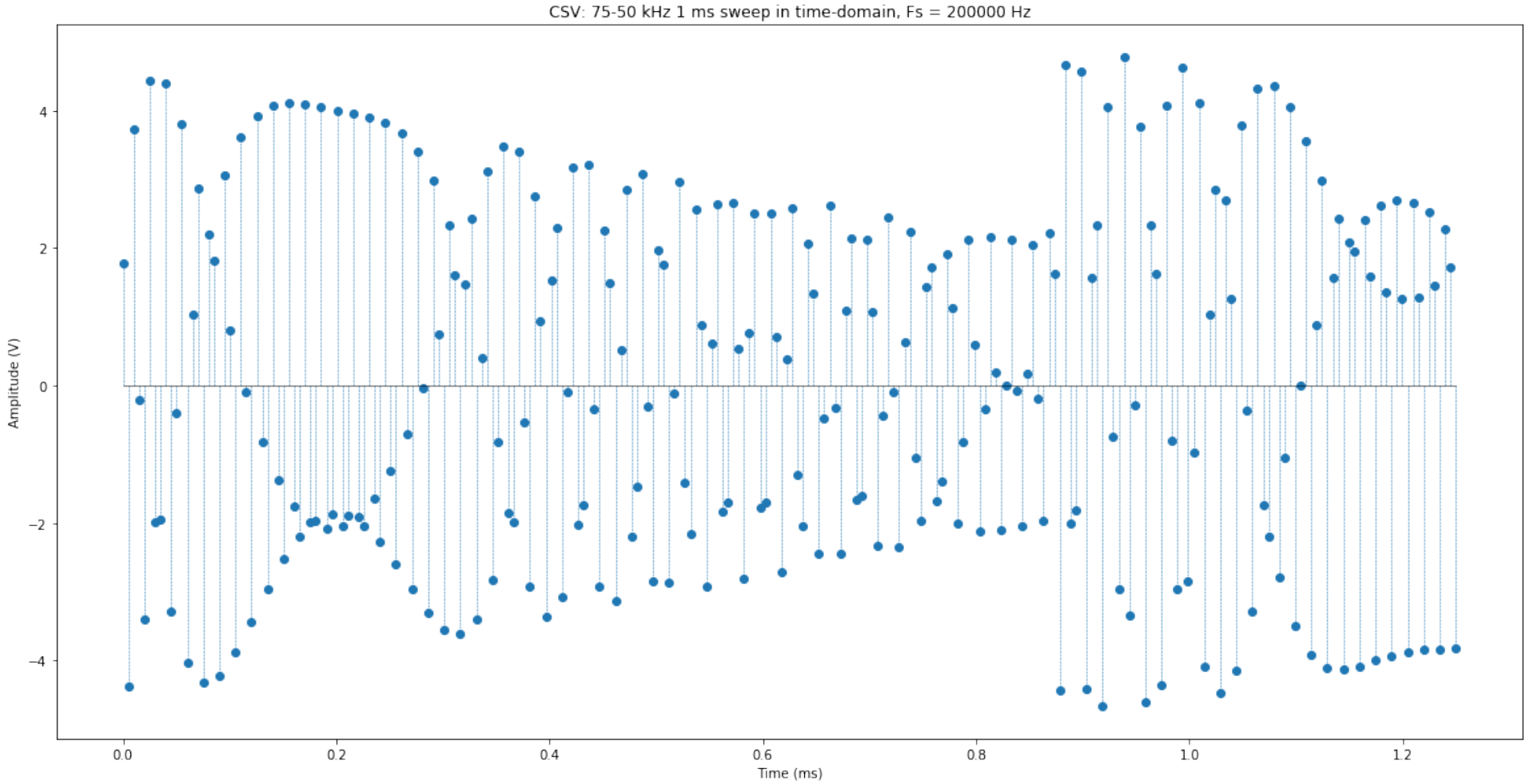
1) 96 kHz 5 V Sine
   - Recording file: `csv/200/sine/96kHz200_20180909T033007480050Z.csv`
   - Temporal domain recording:



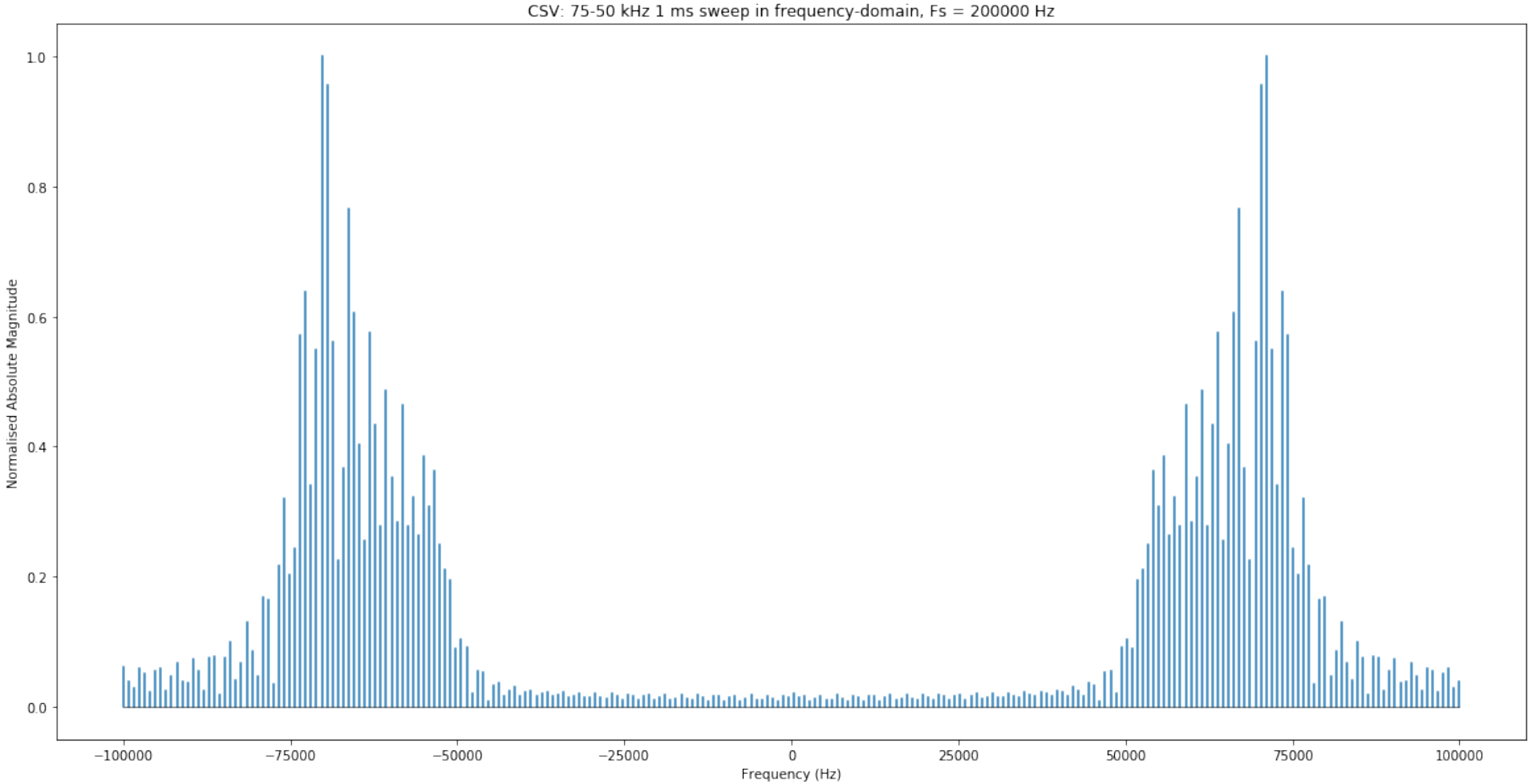
- Spectral domain recording:



2) 75 kHz 5 V to 50 kHz 2 V Sine 1 ms Sweep
   - Recording file: csv/200/sweep/75-50kHz200_20131219T021143244776Z.csv
   - Temporal domain recording:

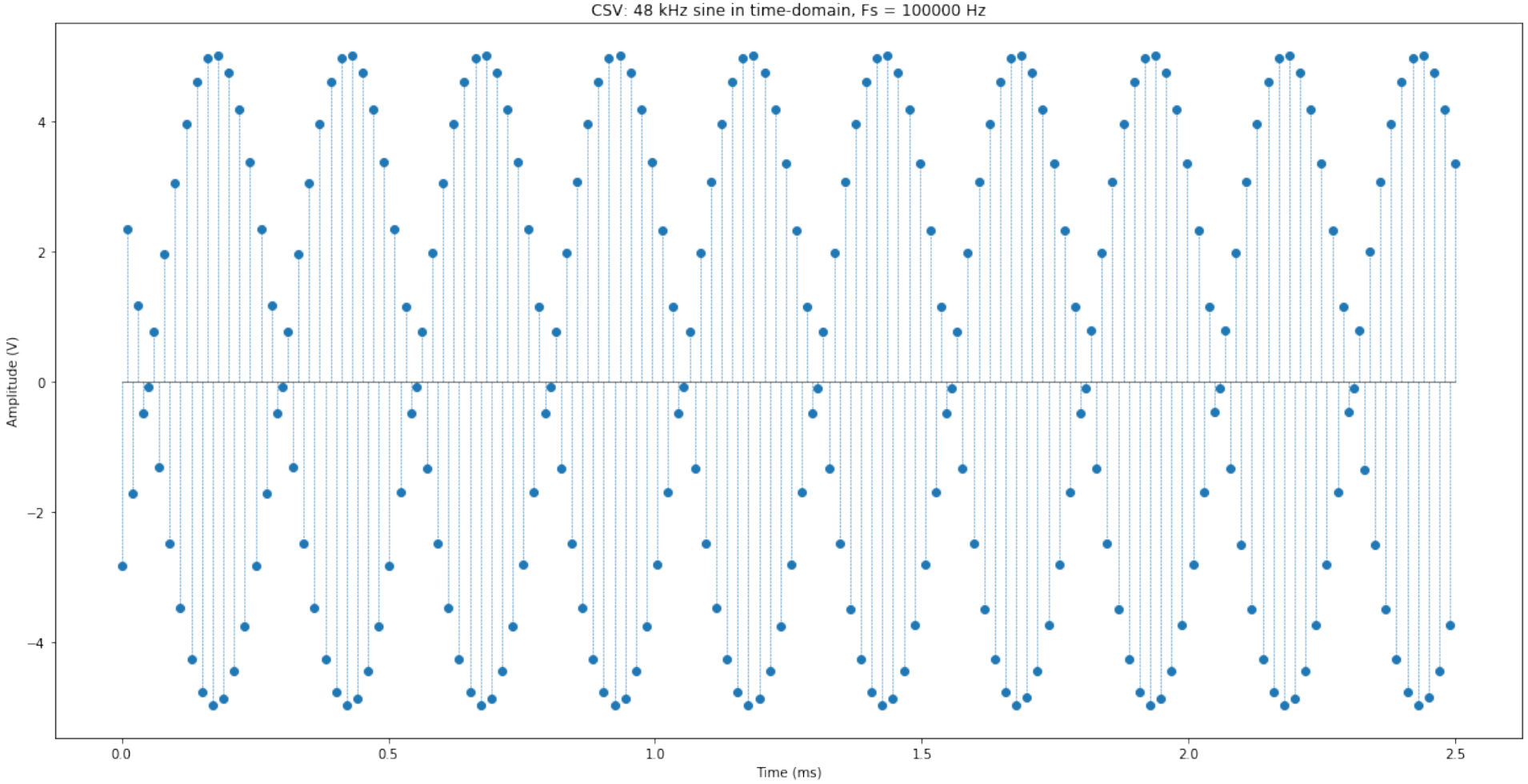


- Spectral domain recording:

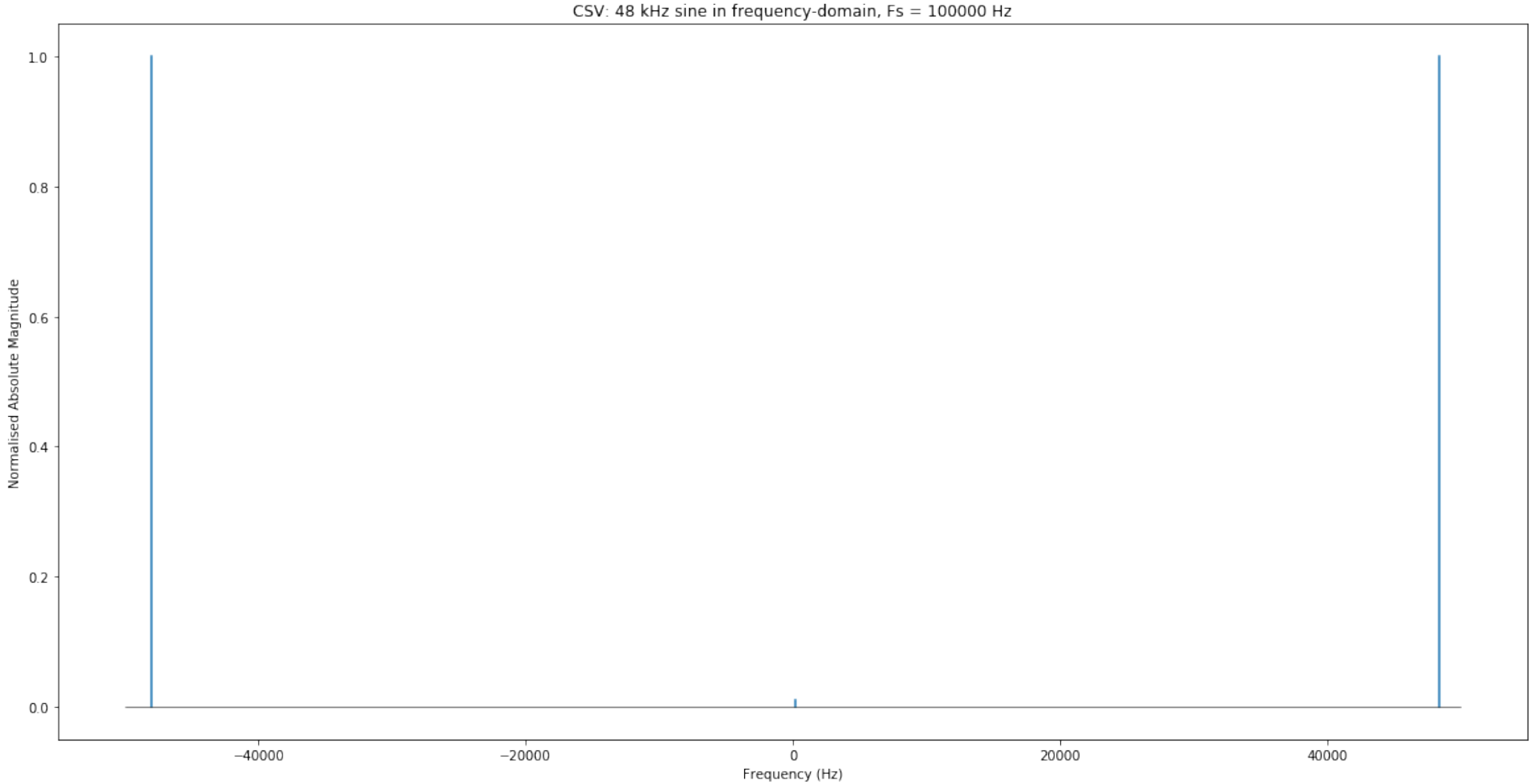


100 kHz

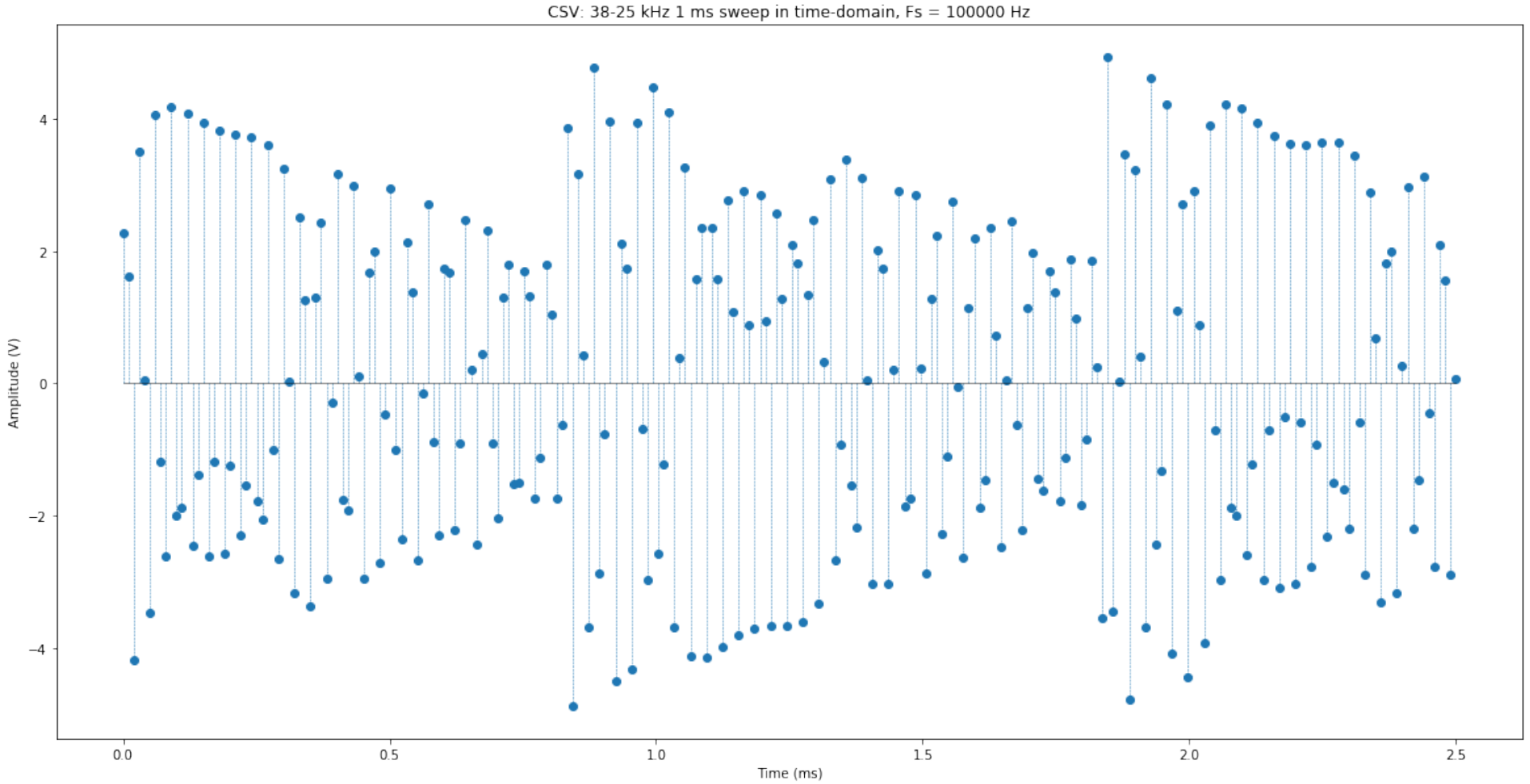
1) 48 kHz 5 V Sine
   - Recording file: `csv/100/sine/48kHz100_20180909T024527672838Z.csv`
   - Temporal domain recording:



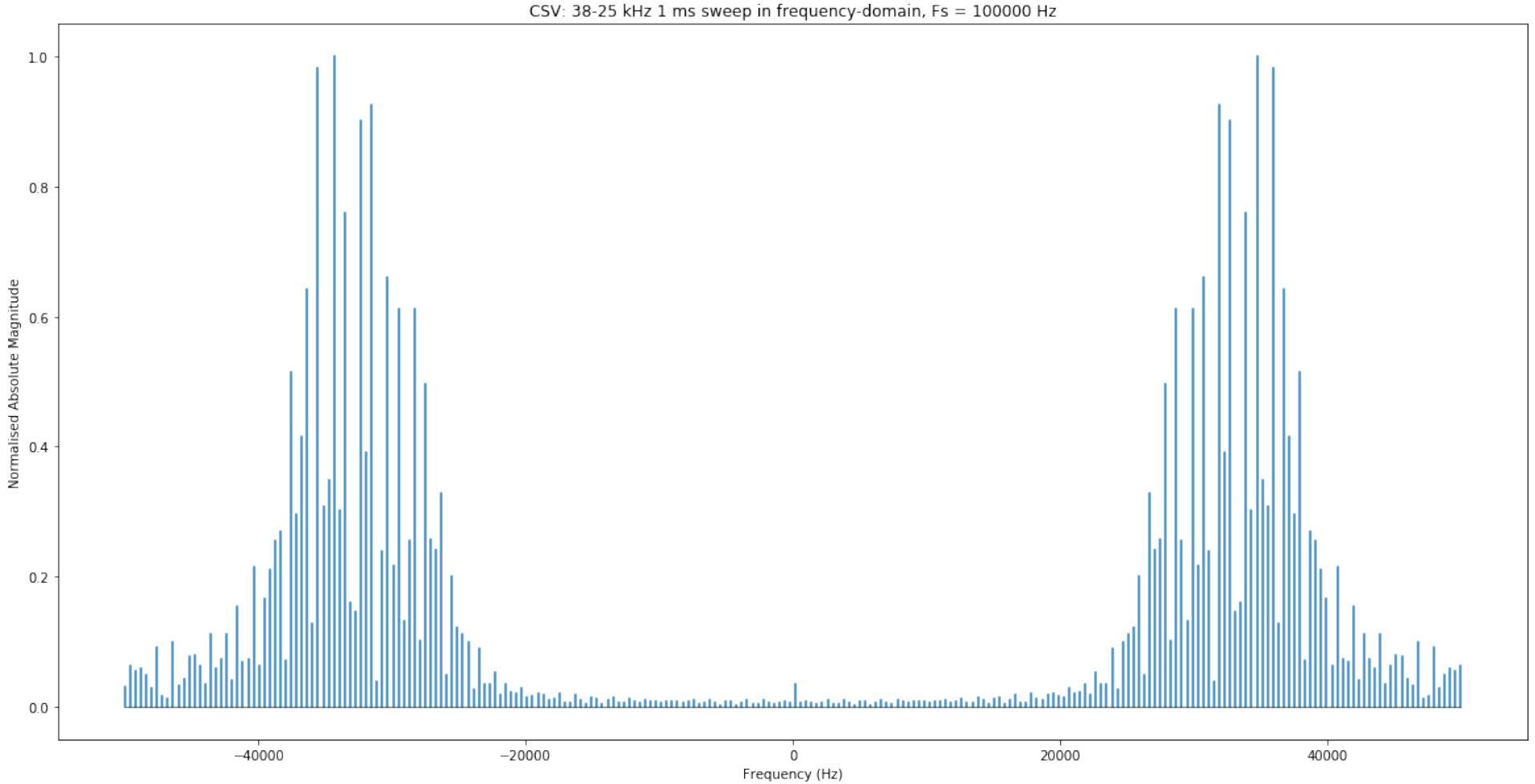
- Spectral domain recording:



2) 38 kHz 5 V to 25 kHz 2 V Sine 1 ms Sweep
   - Recording file: `csv/100/sweep/38-25kHz100_20131219T020906705802.csv`
   - Temporal domain recording:



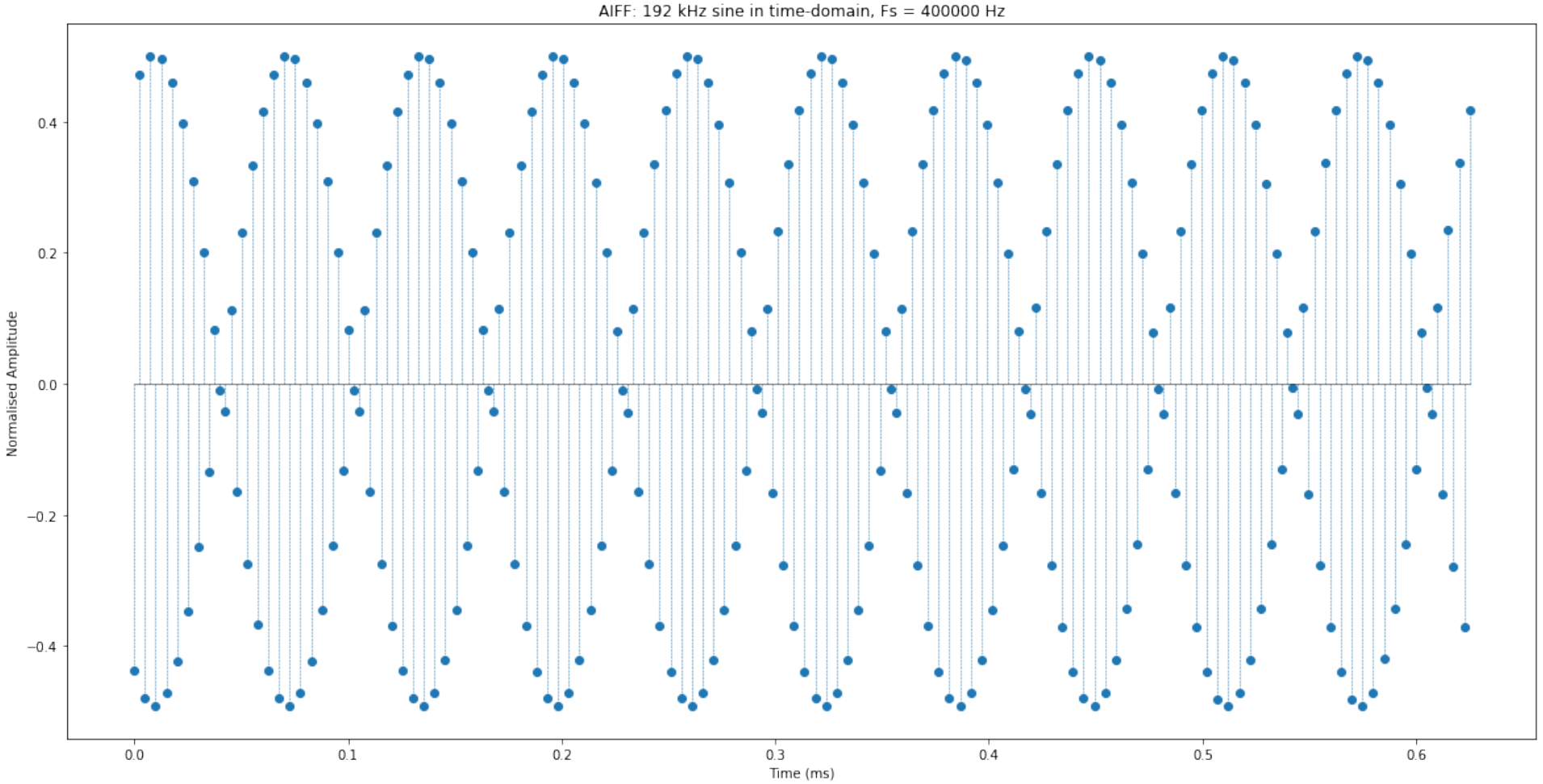
- Spectral domain recording:



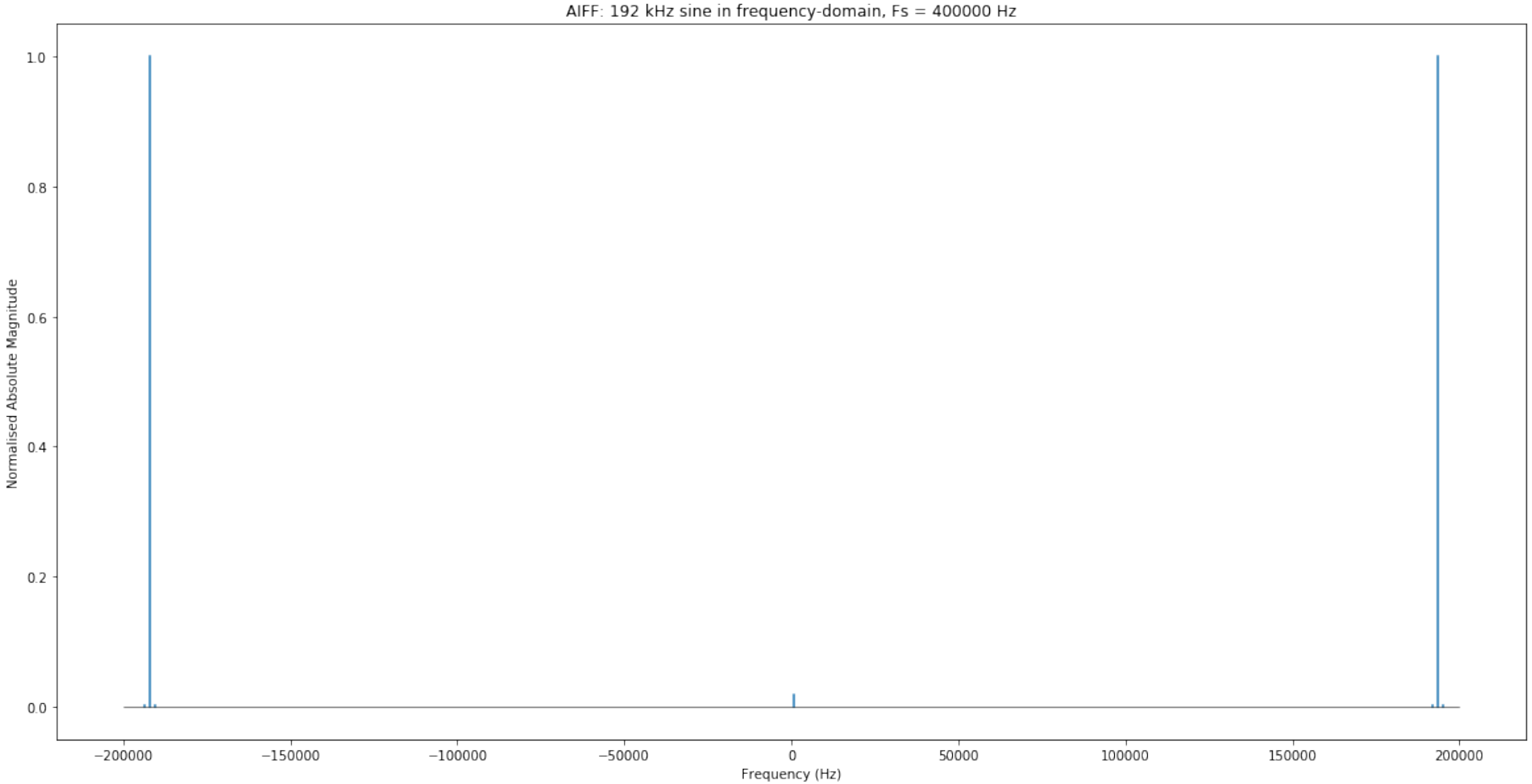
AIFF

400 kHz

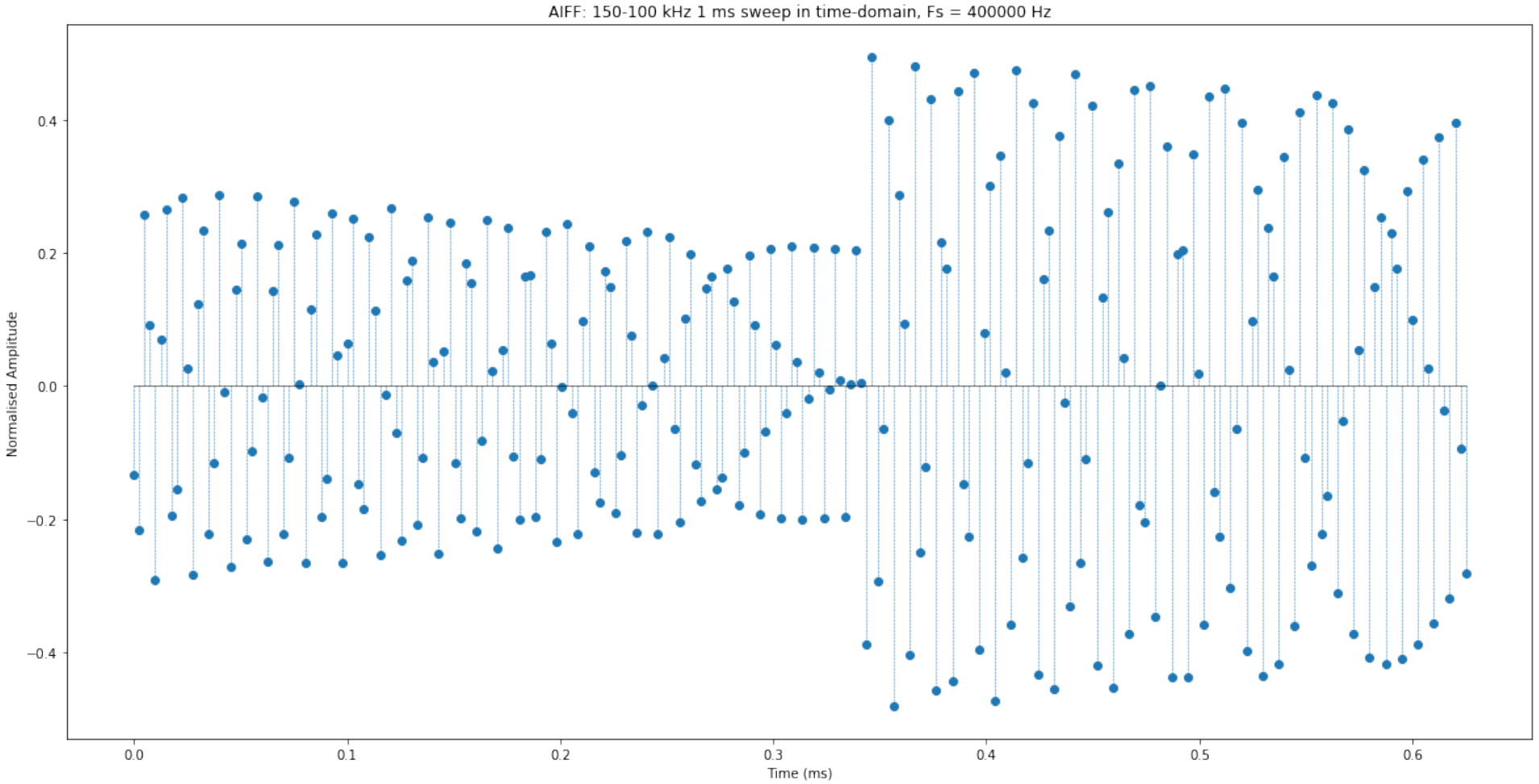
3) 192 kHz 5 V Sine
   - Recording file: `aiff/400/sine/192kHz400_20180911T063429741633Z.aiff`
   - Temporal domain recording:



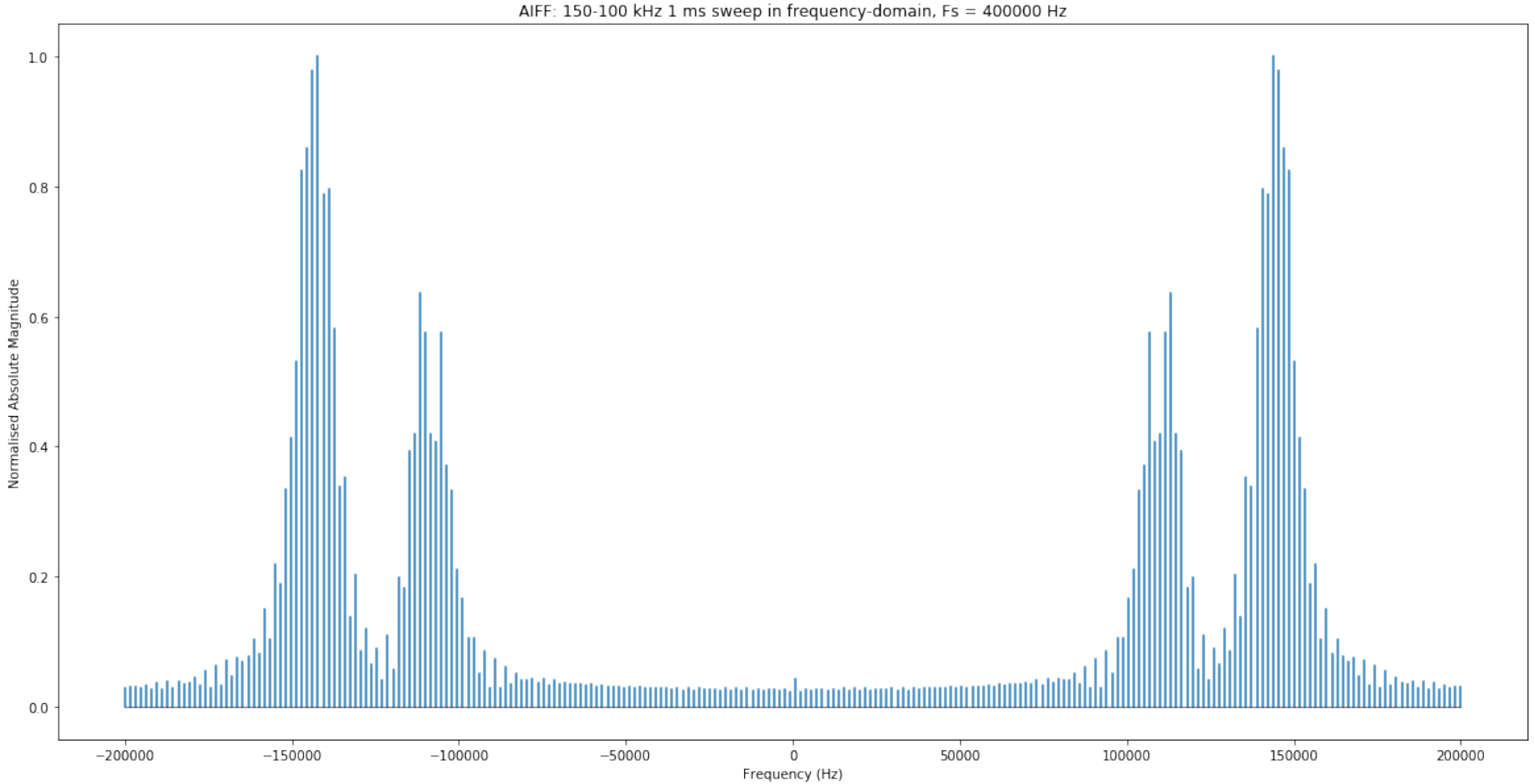
- Spectral domain recording:



4) 150 kHz 5 V to 100 kHz 2 V Sine 1 ms Sweep
   - Recording file: `aiff/400/sweep/150-100kHz400_20180911T0653024680Z.aiff`
   - Temporal domain recording:



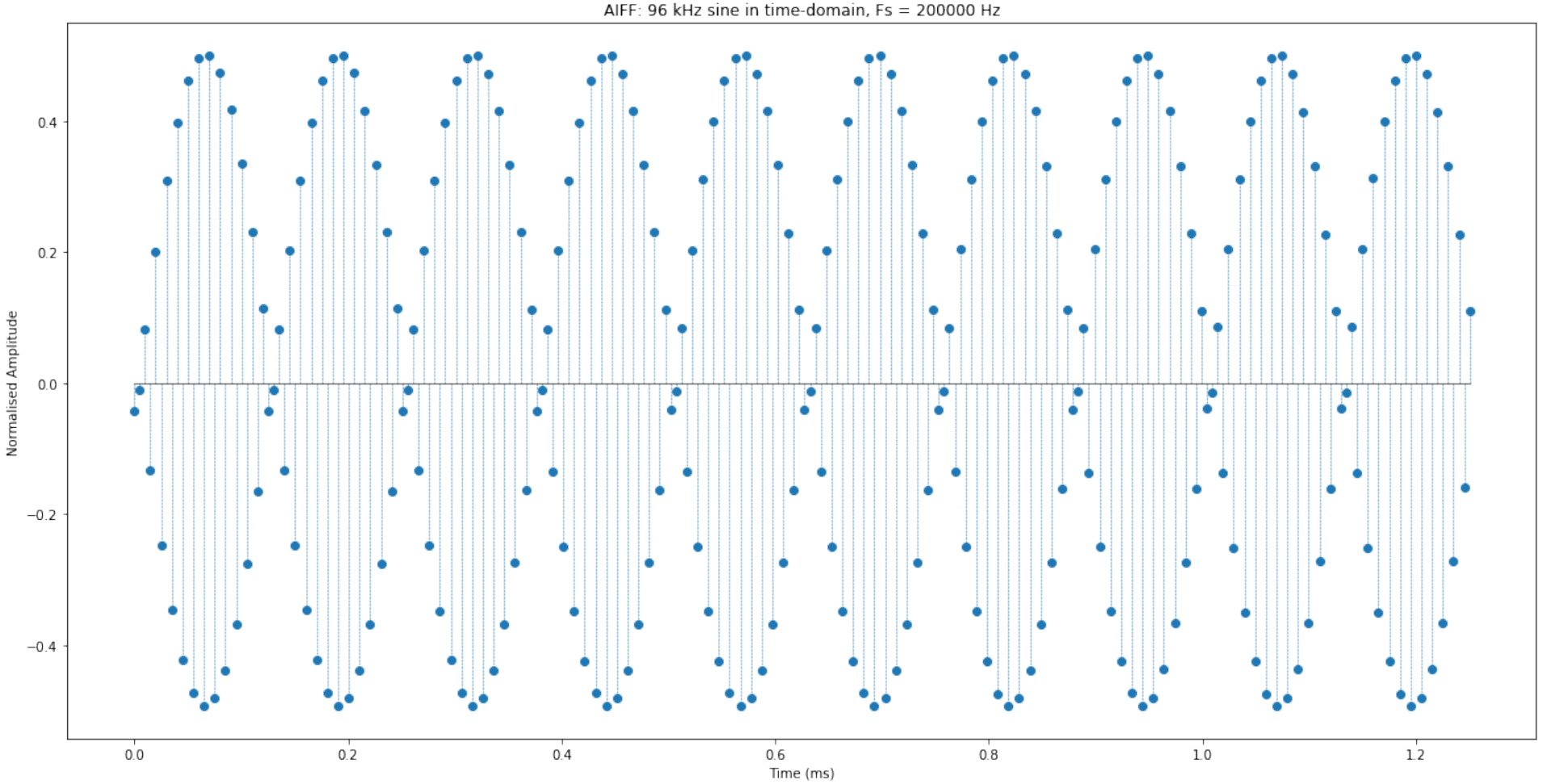
- Spectral domain recording:



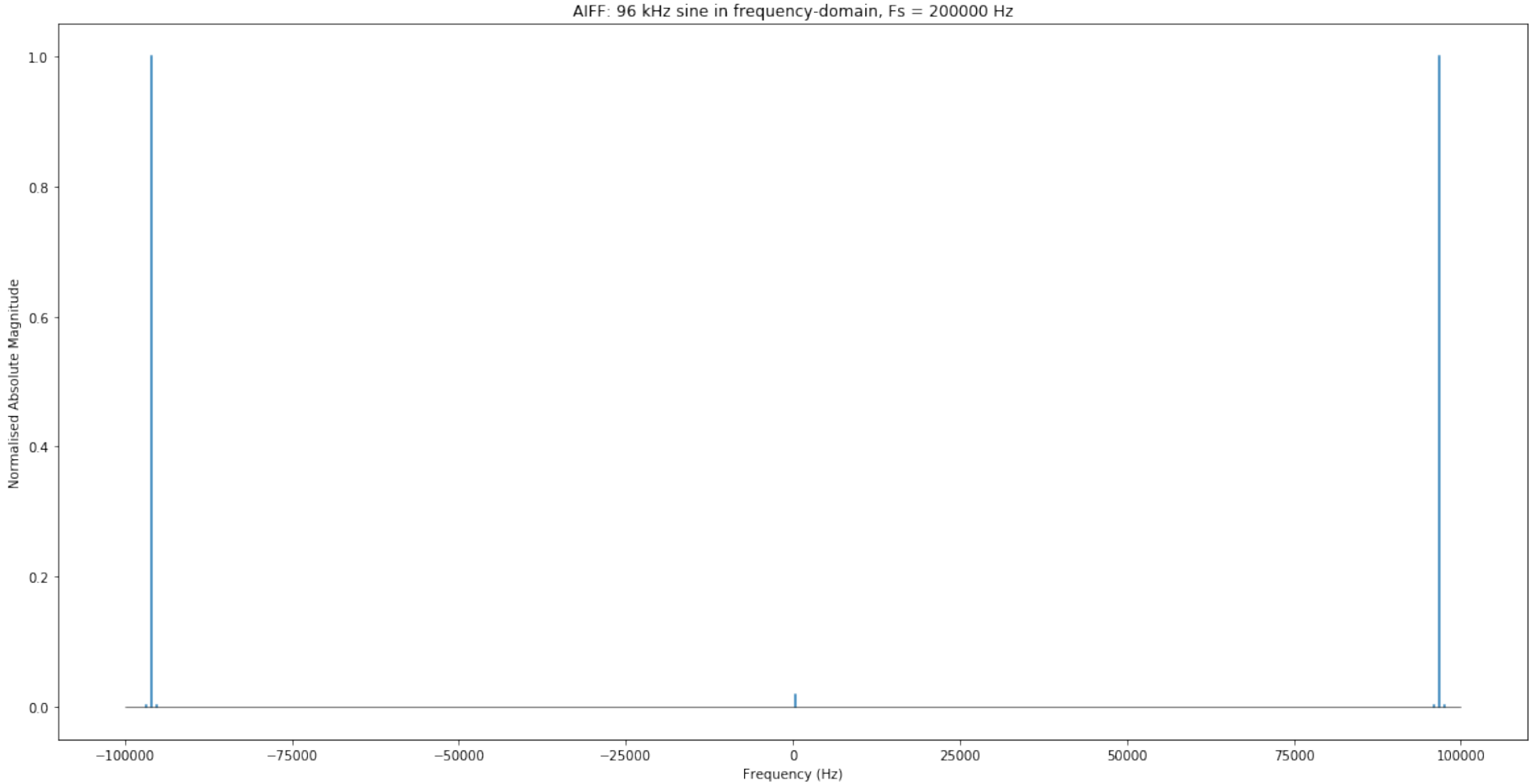
200 kHz

3) 96 kHz 5 V Sine

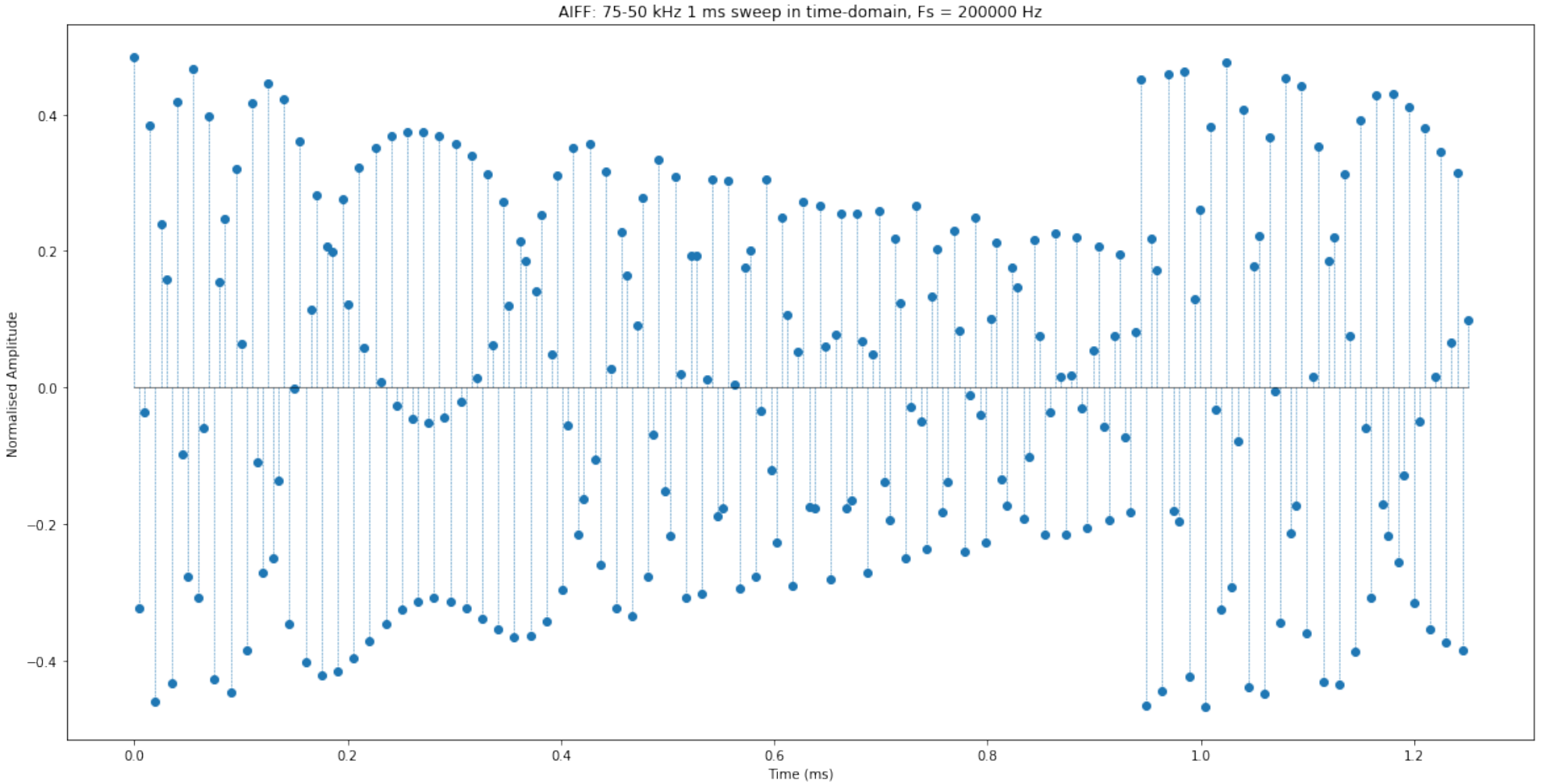
- Recording file: `aiff/200/sine/96kHz200_20180911T062911695433Z.aiff`
- Temporal domain recording:



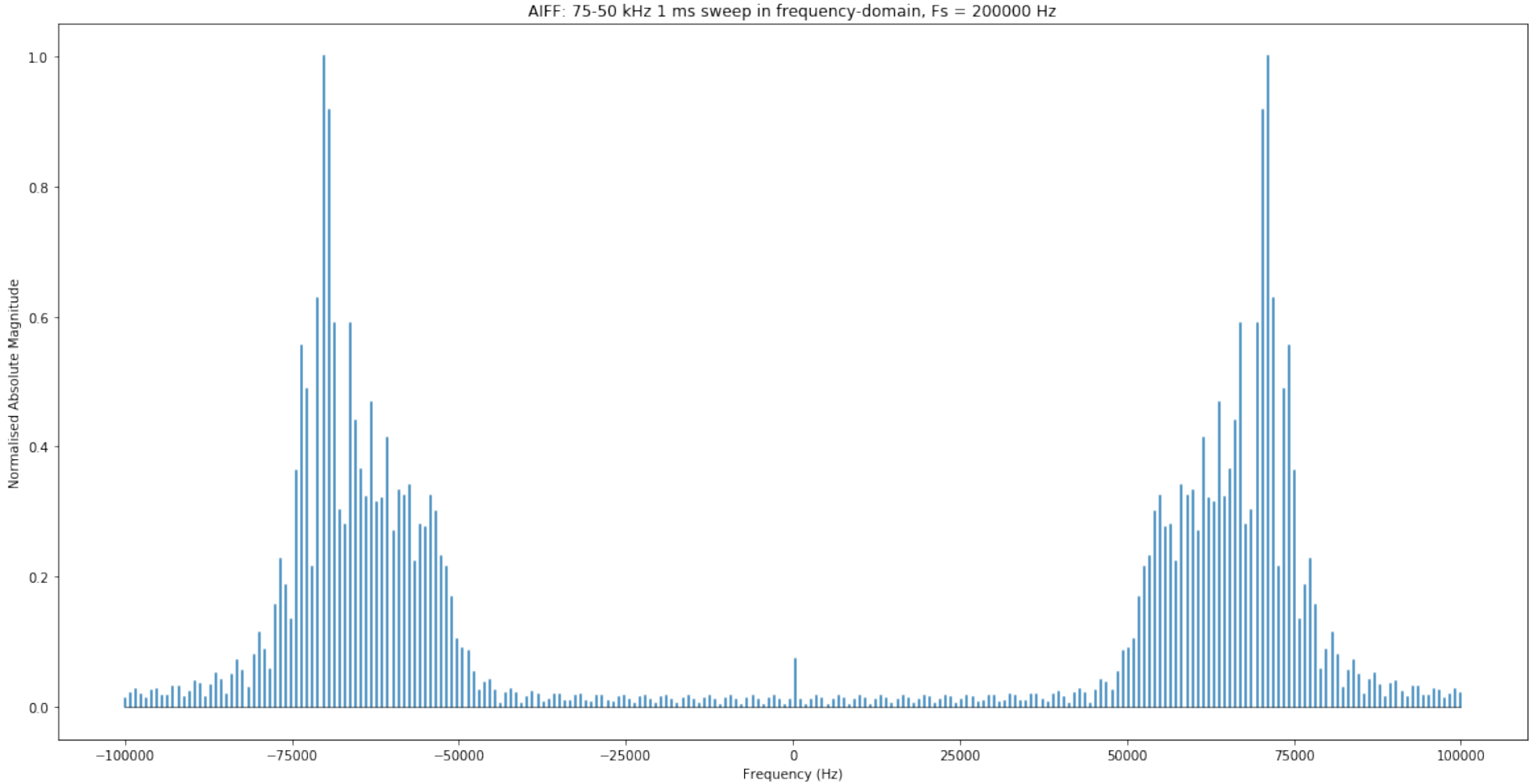
- Spectral domain recording:



4) 75 kHz 5 V to 50 kHz 2 V Sine 1 ms Sweep
   - Recording file: `aiff/200/sweep/75-50kHz200_20180911T064630568199Z.aiff`
   - Temporal domain recording:



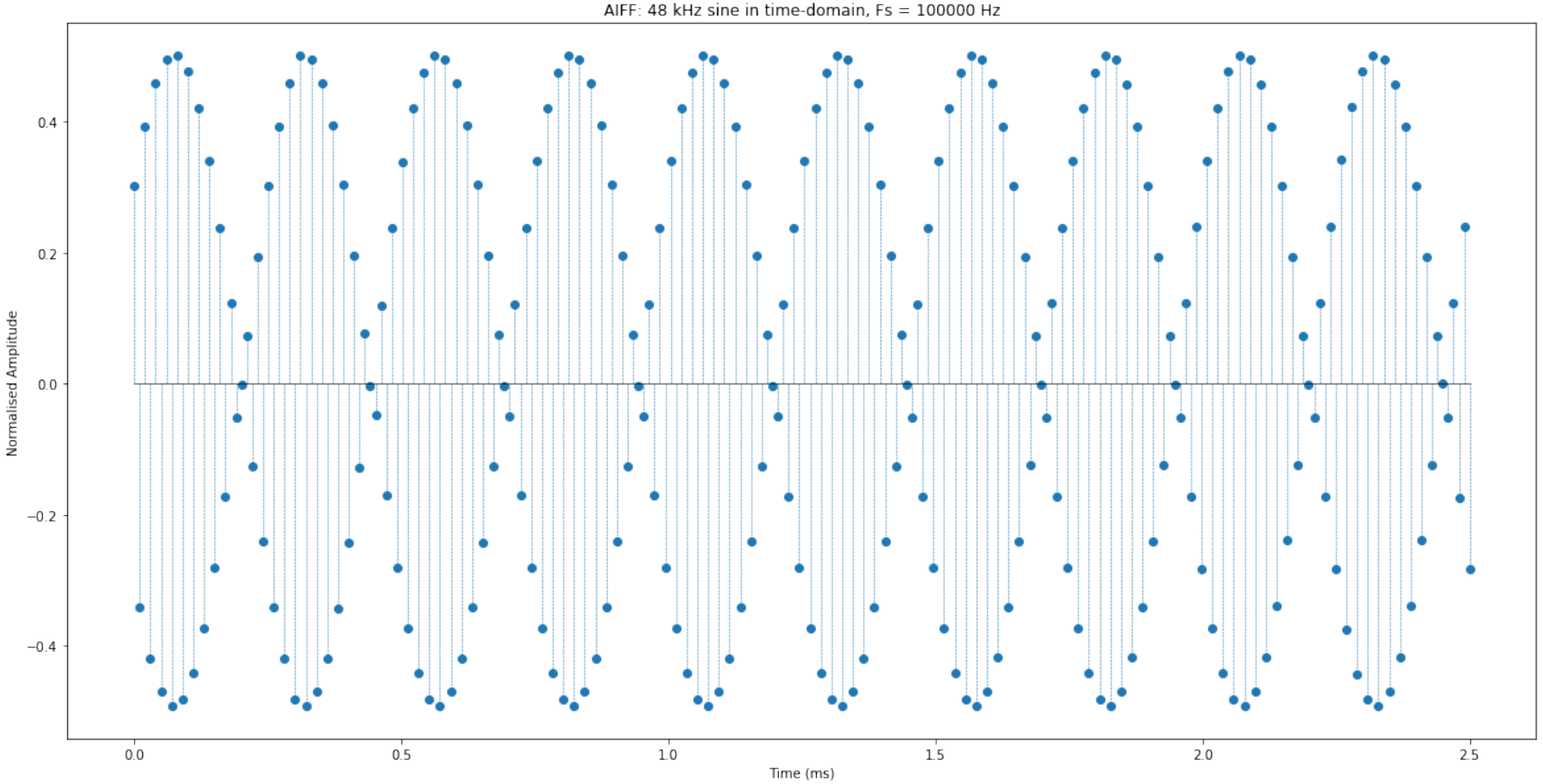
- Spectral domain recording:



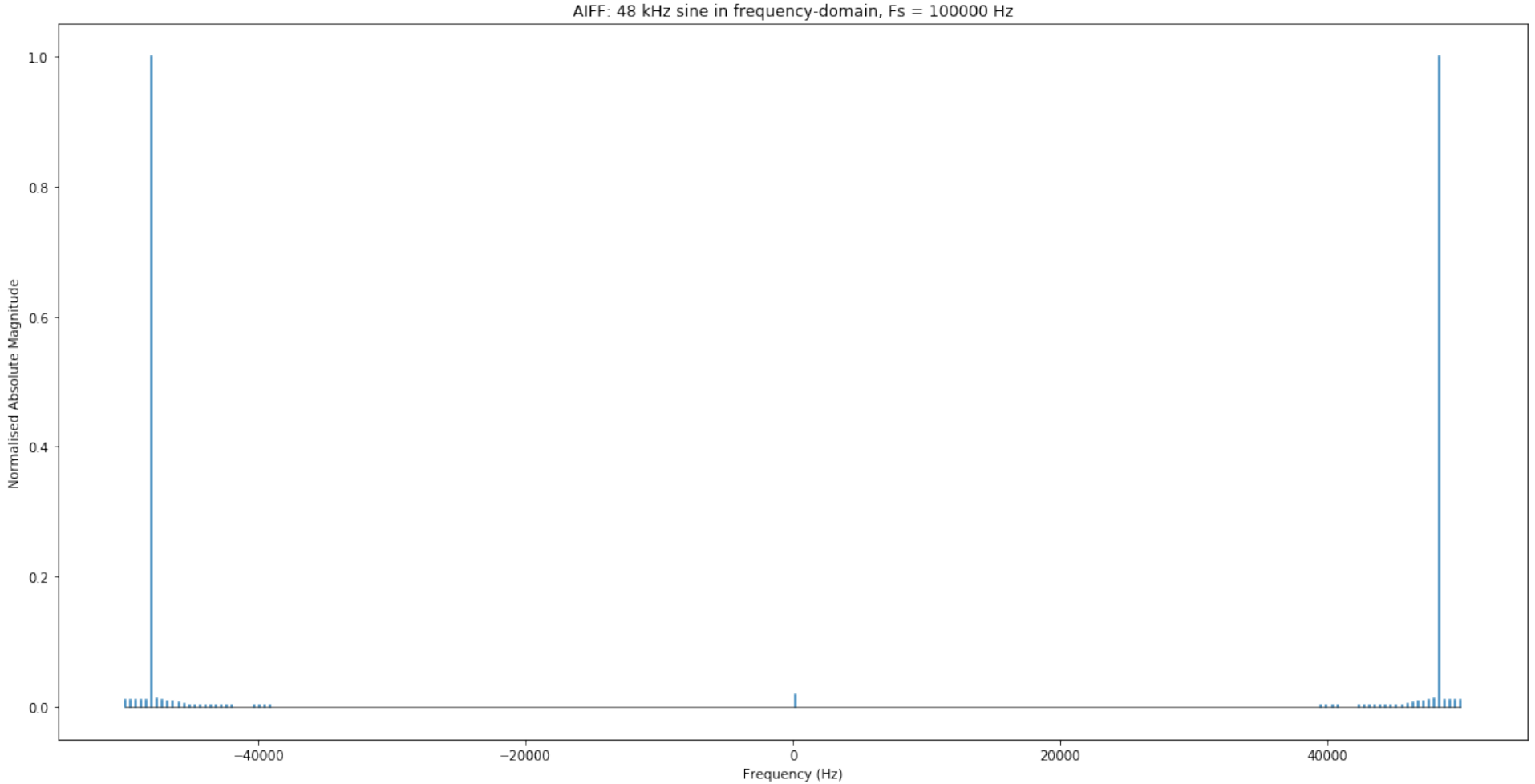
100 kHz

3) 48 kHz 5 V Sine

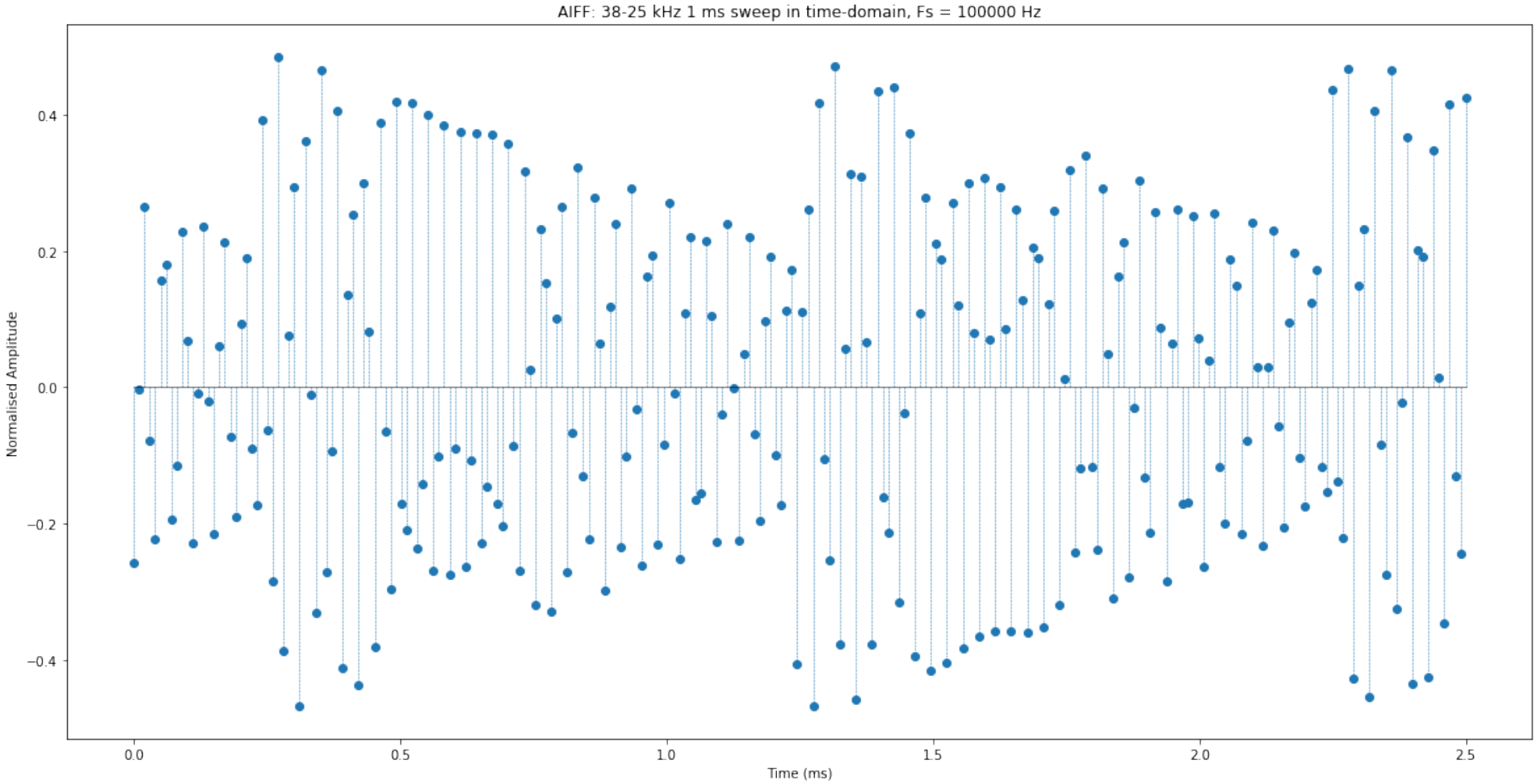
- Recording file: `aiff/100/sine/48kHz100_20180911T062226700725Z.aiff`
- Temporal domain recording:



- Spectral domain recording:



4) 38 kHz 5 V to 25 kHz 2 V Sine 1 ms Sweep
   - Recording file: `aiff/100/sweep/38-25kHz100_20180911T064323844852.aiff`
   - Temporal domain recording:



- Spectral domain recording: