The Sound of the Sun

An Introduction to the Sonification of Solar Harmonics (SoSH) Project

A more extensive discussion of helioseismology with added graphics can be found at http://solar-center.stanford.edu/SoSH/ . Short instructions can be found in `quickstart_audio.txt` . An accompanying python package that provides both visualizations and utility functions referred to in this document is described in `instructions_visual.pdf` .

Contents

Section 1: Background

The Sun is a resonant cavity for very low frequency acoustic waves, and just like a musical instrument, it supports a number of oscillation modes, also commonly known has harmonics. We are able to measure the frequencies of these various harmonics by looking at how the Sun's surface oscillates in response to them. Then, just as the frequency of a plucked guitar string gets higher with more tension and lower with greater thickness, we are able to infer properties of the solar interior such as its pressure and density.

This study of acoustic waves inside the Sun is called helioseismology; an overview can be found at http://soi.stanford.edu/results/heliowhat.html . Although it has allowed us to make measurements of unprecedented precision, it remains largely unknown to the general public. Of course, when one learns of it for the first time, a natural question arises: "What does the Sun sound like?". And unfortunately even helioseismologists rarely have much experiential knowledge of it. That is, we analyze solar data scientifically, but we never listen to it. It is the goal of this project to make such listening widely available.

The first widely recognized effort toward the sonification of helioseismic data was undertaken by Alexander Kosovichev, based on earlier work by Douglas Gough (see http://soi.stanford.edu/results/sounds.html). Although only a small amount of data was sonified, physical effects such as scattering off the solar core were still audible. With the sonification of a vastly larger dataset, one would also be able to hear solar rotation, or perhaps even the effect of the solar cycle. Not only does this bring helioseismology and solar physics into the realm of everyday experience for the nonscientist, but it might even allow for new scientific discoveries. The fact is that we simply don't know what might be audible in the data because we have never listened to it.

Finally, we said above that the Sun is like a musical instrument, but one could also say that the Sun *is* a musical instrument, in that it has its own distinct set of harmonics. Of course we can't play the Sun in the sense of sounding individual notes; rather all of the solar notes are playing all the time simultaneously. With a little analysis, however, the various solar tones can be separated from each other and then used in musical composition.

Section 2: Helioseismology and Sonification

The strongest of the Sun's harmonics have periods of about 5 minutes, corresponding to frequencies of only about 0.003 hertz. Unfortunately, this is far below the range of human hearing, which is typically taken to be 20 - 20,000 hertz, although most people are only sensitive to a smaller range. Hence, if we would like to experience the sound of the Sun with our ears, these very low sounds must be scaled to the range we can hear.

But first, we need some exposition of what a mode on the Sun looks like. To begin with, it is a mathematical theorem that any arbitrary shape of the Sun's surface can be expressed as a sum over its harmonics (this is also true for a guitar string). In the case of the Sun, we call them spherical harmonics, and each of them are labelled by two integers: the spherical harmonic degree l and the azimuthal order m. The degree l is equal to or greater than zero, and for each l, there are $2l+1$ values of m, ranging from $-l$ to l.

The different spherical harmonics also sample different regions of the Sun. Low values of the degree l penetrate almost all the way to the core, whereas higher values are trapped closer to the surface. Similarly, modes with high values of m have their maximum amplitude at low latitudes, whereas lower values sample higher latitudes. It is because the different modes sample different regions that we are able to use their frequencies to determine solar properties as a function of both depth and latitude.

To determine the frequencies of the Sun's harmonics, we might take an image once a minute for 72 days. For each image, we decompose it into its various spherical harmonic components. For each of these components, we form a timeseries of its amplitude. From the timeseries we are able to construct the power spectrum (acoustic power as a function of frequency). The location of peaks in the power spectrum correspond to the frequencies of the modes (harmonics). The height of the peak tells us the mode amplitude, and the width of the peak tells us how much the oscillation is damped.

An easy way to understand spherical harmonics is in terms of their node lines, which are the places on the sphere where the spherical harmonics are zero. The degree l tells how many of these node lines there are in total, and the order m gives the number in longitude, so the number of node lines in latitude is l-m. So a spherical harmonic with m=0 has only latitudinal bands, while one with m=l has only sections like an orange. A third integer, the radial order n, tells how many nodes the oscillation has along the Sun's radius. Since only the surface of the Sun is visible to us, all the values of n are present in each spherical harmonic labelled by l and m, although only some of them will be excited to any appreciable amplitude. The total mode, then, is represented as a product of a spherical harmonic, which is a function of latitude and longitude, and another function of radius, which depends on both n and l. Each n will have its own peak in the power spectrum.

Now, in a spherically symmetric Sun, all values of m for a given l and n would have the same frequency. A break in spherical symmetry causes the frequency to vary with m. The most significant deviation from spherical symmetry in the Sun is rotation about its axis. The spherical harmonic decomposition, however, is only sensitive to the absolute value of m. Therefore the positive and negative values of m must be separated in the power spectrum. We say that the positive frequency part of the spectrum corresponds to positive m, and that the conjugate of the negative frequency part corresponds to negative m. Note that this particular convention for the sign of m is completely arbitrary. If you want to understand what is meant by "the negative frequency part of the power spectrum", you will need to study the Fourier transform, but such understanding is not strictly necessary.

Let us now return to the issue of sonification, the conversion of data into audible sound. The most straightforward way to do so would be to use the spherical harmonic timeseries we already have in hand and speed them up. But by how much? The answer of course is arbitrary and will depend on your preference, but as long as this choice is applied consistently you will still be able to hear the real relationship between different solar tones.

Let us suppose that we want to transpose a mode in the peak power range at about 0.003 hertz up to 300 hertz; this amounts to speeding up the timeseries by a factor of 100,000. If we have 72 days of data taken once a minute, this amounts to 103,680 data points, or samples. It is easy to see that the sped-up timeseries would now play in just over a minute. One must also consider the sample rate, however, or the rate at which audio is played back. Speeding up the original sample rate of (1/60) hertz by a factor of 100,000 yields a new sample rate of 1666.67 hertz, and one unfortunately finds very few audio players that will play any sample rate less than 8000 hertz. Assuming this sample rate, our 0.003 hertz mode on the Sun will now be transposed up to 1440 hertz and the timeseries will play in about 13 seconds.

But suppose you want to play it in a shorter time; 13 seconds is a long time to sound a single note, although you might want to do so in some circumstances. You could increase the sample rate further still, but at some point the mode will be transposed to an uncomfortably high frequency. To understand the solution to this problem, we must explore the process by which we shall isolate the modes.

At this point in our processing, playing an unfiltered timeseries would sound just like static, or noise. This is because very many modes are sounding simultaneously in any given timeseries, not to mention the background noise involved in our observation of the modes. Therefore, if we want to isolate a single mode, we have to do some filtering. Luckily, as mentioned above, we have already measured the frequency, amplitude, and width of many modes. We can use these fitted mode parameters to pick out the particular part of the power spectrum corresponding to a single mode, and set the rest of the power spectrum artificially to zero. We then transform back into a function of time so that we can play the filtered data back as a timeseries.

Since we are selecting only a narrow range of frequencies, we have the freedom to shift the entire power spectrum down in frequency before we transform back to timeseries. This timeseries will play in the same amount of time as before, but the frequencies in it will be transposed down by the same factor that we shifted the power spectrum. For every power of 2 shifted down, the tone will drop by one octave.

One approach might be to decide how long you want to sound each tone (keeping in mind that looping is also an option). This will determine the sample rate at which you will play the timeseries. Then you can choose a downshift factor to suite yourself. As long as you use the same sample rate and downshift factor when you sonify every mode, the frequency relationships between them will be preserved.

Section 3: Data

In order to use the Sonification of Solar Harmonics (SoSH) tool, you will first need to download some data. Two types of data are required: text files containing ascii tables of fitted mode parameters and wav files containing the raw acoustic data. Furthermore, these data can originate from two different instruments: the Michelson Dopper Imager (MDI) and the Helioseismic and Magnetic Imager (HMI). MDI is the older instrument, and took data from May 1996 to April 2011. It was superseded by HMI, which began taking data in April 2010 and remains in operation today. The two instruments are quite similar; the most important difference between them for our purpose is that MDI produced an image once a minute, while HMI produces an image every 45 seconds. For both instruments, however, we analyze the data using 72 day long timeseries.

If you have also downloaded the SoshPy Visualization Package, then you may use its included python module to interactively retrieve whatever data you need. Instructions for this module are included in the package. However, these data may also be downloaded from the web, and instructions for doing so follow.

First, pick a single directory for storing data on your computer; for your convenience we have included a data directory in the zip archive you have unpacked. If you elect to use a different directory (such as your browser's download directory), you will simply need to enter it the first time you run the SoSH tool. If you have downloaded the version of the SoSH tool that includes demo data files, these will already be found in the unpacked data directory. The quickstart guide includes instructions for using those specific files.

In any case, the data are available at http://sun.stanford.edu/~tplarson/audio/ , where you will find separate directories for MDI and HMI. Within each, you will find a series of subdirectories that are day numbers suffixed with 'd'. The day number corresponds to the first day of the 72 day timeseries and is actually the number of days since 1 January 1993. MDI began taking data on day number 1216, which was 1 May 1996. HMI began taking data on day number 6328, which was 30 April 2010. A full table converting day numbers to dates can be found at the above url as well.

Clicking on a directory will show two ascii files containing the mode parameters; download both of these to your data directory. The file without "msplit" at the end contains one line for every degree l and (radial) order n for which the fitting succeeded. The first five numbers of each line are all that we will use here; they are degree l, order n, mean frequency, amplitude, and width. These numbers are the same for all m, the mean frequency being the average over m. The file with "mpslit" at the end tells us only how the frequency varies with m. Make sure that however you download them, the file names stay intact; some browsers like to add or delete filename extensions. This is good to check if you get "file not found" errors later.

UPDATE: the "msplit" file is no longer required by default.

You need not view these files, but keep in mind that in general the fitting does not succeed for every value of n. Put another way, every file will have l values ranging exactly from 0 to 300, the upper limit being a somewhat arbitrary choice. For every value of l, exactly $2l$+1 values of m will appear in the msplit file. What may vary widely, however, is which values of n appear for different values of l. We typically only find modes with n=0 in timeseries with l>100. Modes with n=28 (higher values are rare) are only likely to be found for l=5-15. Looked at from the other direction, for l=0 we typically find modes with n=10-25. For l=100, n=10 is the highest value we might find. Above l=200, we have only n=0. In all cases, one may expect to find holes in the coverage of n for values of l close to the edge of the range.

One way to get much higher mode coverage, at the cost of time resolution, is to perform an average. An example of such an average can be found in the files `mdi.average.modes` and `hmi.average.modes` in the corresponding directories at http://sun.stanford.edu/~tplarson/audio/, along with the corresponding msplit files. The stand alone patch described in the next section is set to use these averaged mode parameters by default, which means those averages will be used for all day numbers. In this case, no further mode parameter files would need to be downloaded.

Next you will need to download some actual audio files. To do so click on the wavfiles subdirectory, where you will find a selection of modes, labelled by l and m. Each file contains both signs of m (recall that these must be separated in the frequency domain). Except for m=0, each mode has both a real and an imaginary part, labelled by "datar" and "datai" respectively; make sure you always get both. These files contain exactly the same data as we use for scientific analysis. The file formats have simply been changed from fits (flexible image transport system) to wav. Pick an assortment of modes and download them to your data directory. Of course you may play these sound files just as they are if you want to hear the unfiltered data.

Note that only a subset of the spherical harmonic timeseries are available as described above. By using the tools included with the SoshPy Visualization Package, you may retrieve and convert any or all available spherical harmonic timeseries.

UPDATE: by starting a certain python process in the background before running the SoSH tool, data can now be downloaded automatically from within Pure Data.

Section 4: Software (Pure Data)

Now we are ready to dive into a description of the software by which the scheme laid out in section 2 can be accomplished. One freely available option is Pure Data, available from http://puredata.info . Pure Data provides a graphical interface for audio processing. Programs in Pure Data are called "patches". Once you have it installed, run the program and the main Pd console will open. First you will want to test that it is communicating with your soundcard. To do so, click on "Media" and then "Test Audio and MIDI". This will open a patch called `testtone.pd` . If you see changing numbers under audio input, you are connected to your computer's microphone, although that is unneeded for this project. More important is the audio out, which you can test by clicking in the boxes under "test tones".

Once this is working, you are ready to use the SoSH tool. Unzip the archive and open the patch `modefilter_cleaned.pd`[1]. If you've never looked at a Pure Data patch before, this will probably look rather confusing, so i will provide an extremely brief introduction. There are three types of boxes in Pure Data: messages, numbers, and objects. Messages are the boxes with an indentation along the right side, perhaps to make the box look like a flag. Messages are basically the equivalent of strings, but they can also be automatically converted to numbers. Number boxes have a little notch out of the upper right corner. The internal storage for numbers is floating point, but you can also cast to int. An important difference between number boxes and message boxes is that the contents of the latter can be saved. For instance, if one wants to initialize a number, a common way is with a message. Also, numbers may be entered while the patch is running, whereas messages cannot. The remaining rectangular boxes are objects, which are like functions. The first element in an object box is the name of the object, which often corresponds to a patch file (extension `.pd`) of the same name. This is optionally followed by the object's creation arguments. All three types of boxes have inlets on the top and outlets on the bottom.

Another important concept in Pure Data is its own unique data type called bang, which can be thought of like a mouse click. The message "bang" will also automatically convert to a bang. Bangs are used throughout Pure Data as triggers for various events, or they can be used to signal event detection as well. In the graphical interface, bangs are represented by clickable circles, which we have enlarged and colored green or light blue in our patch. You have probably noticed an object called `[loadbang]`; its sole purpose is to output a single bang when the patch loads. This is typically used for initialization: sending a bang to a number or message box causes its contents to be output. You may also notice a toggle, represented as an empty green square in our patch, also clickable. This functions like a normal boolean, but it is not a separate data type; it is simply 0 or 1. Finally, arrays in Pure Data, also called tables, come with their own graphical representation. Examples visible on the front of our patch are `gain`, `input-r`, `input-i`, and `output`. (The `$0` preceding these names resolves to a unique integer when the patch loads; this becomes necessary when this patch is used as a subpatch to avoid conflicting names.

1 You may elect to open `modefilter_standalone.pd` instead, in which all connections are shown explicitly. Unfortunately this makes the patch rather cluttered, which is why we created `modefilter_cleaned.pd` for exposition.

Other dollar sign substitutions are more like one would expect: they resolve to some element of the input, depending on the context.)

Now, to use the patch, the first thing you have to do is make sure the data directory is set properly. If you are using the data directory that was unpacked along with the zip archive, you don't have to do anything, since the patch is already to set to look in `../data`. Otherwise, set the data directory by clicking the light blue bang at lower left. A dialog box will open; just select any file in your data directory and the object [set-directory] will strip the file name and output the path. You should now see your path show up in the message box at right. If you now save the patch file, this will be saved as your default data directory and you won't need to set it any more.

By default, the patch is set to use MDI data. In particular, this means that it assumes the data was taken at a sample rate of (1/60) hertz, which in turn means that 72 days of data contain 103680 data points. The patch will also use the string "mdi" as the stem for input file names. If you are using HMI data, you may click the message box with "hmi" at the lower left of the patch, and this will be used as the stem for file names. The HMI sample rate of (1/45) hertz will also be used, which means that 72 days of data would contain 138240 data points.

The next step is to click on the message box with "pd dsp 1", which will turn on digital signal processing (you can also do this from the Pd console). This doesn't need to be on to load files or access arrays, but it does to do any Fourier transforms or playback. Finally, the inputs you must provide are the day number corresponding to the 72 day timeseries, the spherical harmonic degree l, the radial order n, and the azimuthal order m. Note that even if you want to leave one of these at its default value of zero, you must still click on the number box and enter "0". Now, to search for this mode, click the green bang at the upper left. The object [triggerlogic] will then decide what to do. Typically, the object [loadaudio] is triggered, as you will see when the bang connected to it flashes. We have left these large bangs throughout the patch to make the triggering visible, but you may also use them to manually run the different parts separately.

By default, the patch will look for an averaged mode parameter file. If you have downloaded the mode parameters for a particular day number and wish to use them, you must click on the message box with "%dd.modes" at the bottom left of the patch before clicking the bang to start processing. The "%d" will be replaced with the day number you have entered.

The object [loadaudio] searches for audio files such as the ones you have just downloaded. Note that the needed input files will depend only on l and the absolute value of m, and that the modes have a real and an imaginary part. The exception is m=0, which has a real part only. If [loadaudio] is successfully able to load the necessary audio files into the arrays input-r and input-i, it will automatically trigger the object [fft-analyze]. This object will perform a fast Fourier transform (fft) of the input arrays, storing the result in two arrays called fft-r1 and fft-i1. If you want to see these arrays, simply click on the object [pd fft-arrays]. If you do so you will see an option to write them out to wav files, in case you want to compare Pure Data's fft output to your expectations. You will also see two unused arrays; these could be used to store amplitude or phase information if such were desired.

By this time, if you are inquisitive enough, you might have noticed that (for MDI) all of the arrays so far have a length of 131072, which is 2^17, rather than the actual number of samples, which is 103680. This is because Pure Data requires the block size to be a power of 2 for its fft algorithm. The result is effectively to zero pad the end of the timeseries. This has no effect on the frequency content of the sound, and we truncate the output array at the original number of samples, so it will play in the same amount of time as the original. (If you to set block size to less than the number of samples, only this many are output.) If using HMI data, the number of samples is 138240, and so a block size of 2^18=262144 will be used.

(Note: Pure Data in windows doesn't work with block sizes above 65536. If you are running windows, you may have already seen Pure Data crash. To avoid this, click on the message box containing "65536" before turning on the dsp. The patch will function normally, but only the first 65536 samples of each file will be used. You may also avoid the crash by altering the source code and recompiling. A discussion of this topic can be found at https://github.com/pure-data/pure-data/issues/68 .)

At this point, if the fft has been performed successfully, the object [text-filer-reader] is triggered. This searches the text files you downloaded earlier to find the mode parameters corresponding to the numbers you entered. If it finds the mode, it ouputs, after its status code, the mode's amplitude, width, and a measure of background noise. If no mode is found, the status code triggers the message "no data found", and you should try another value of n. (Also check the Pd console for error messages.) The amplitude is wired to a number box for your information. The width will be converted into units of bins and then used as input to the object [makegain]. The noise parameter is unused here. These three parameters will depend on l and n, but not m. Finally, the last outlet from [text-file-reader] gives the mode's frequency, which does vary with m. The frequency is also converted to bin number, and the [makegain] object is triggered. This function creates the array gain, which will be 1.0 in a frequency interval centered on the mode frequency and of length 2 times the width, and 0.0 elsewhere. If so desired, you may enter the parameter "width factor" to multiplicatively increase the width of this interval. Notice how a message was used to initialize this number to 2.5, which corresponds to the width originally used for the fitting. (In `modefilter_cleaned.pd` the bin calculations have been hidden in the [pd converttobins] subpatch.)

UPDATE: you now have the option to use the same frequency interval for all m, which is now the default. This is appropriate for small values of m or large values of the width factor. To turn frequency splitting on, click the toggle connected to [s mswtich] at the bottom center of the patch.

Once the gain array is generated, then one of the [fft-resynth-???m] objects will be triggered, depending on the sign of m. As mentioned above, the two signs of m are extracted differently from the Fourier transform, but in both cases the fft is multiplied by the gain array and then inverse transformed, the result being written into the output array ($0-output). If you have entered a value for the downshift factor, the fft will be shifted down by this amount before the inverse transform. Note that we treat a value of zero as meaning no shift.

Next, the output array is played back in a loop. The default sample rate is 8000 hertz, but you may go up to 44100 hertz for the 103680 samples to play in only 2.4 seconds (in that case, if you haven't applied any downshift factor, the mode will sound extremely high). You can change the sample rate by clicking on one of the nearby message boxes, or by entering one manually. To hear the output, you will need to enter the output level (volume). Note that each loop is multiplied by a window function, which consists of a 50 ms fade in/out at the beginning/end of the loop. The length of the fade ramp may be adjusted on the front of the patch in the lower left corner. The object [window-gen] then calculates the array window. In section 6 we discuss how and when you might turn windowing off.

At this point you may adjust the downshift factor, which will retrigger the resynthesis, and the result should play immediately. You can turn off playback by clicking the toggle. You may also elect to save the output as a wav file file by clicking the light blue bang at lower right. The instrument, day number, l, m, and n will be encoded in the output file name.

Now, should you like to listen to another mode, you may enter its "quantum" numbers l, n, and m, and then click on the green bang again. If only n or the sign of m has changed, no new audio needs to be loaded, and the object [text-file-reader] is triggered directly. The rest of the processing chain follows as before. Note that the names of output wav files do not encode the sample rate, downshift factor, or width factor. Hence, if you want to save the same mode with different values for these parameters, you will have to rename the output file or manually edit the patch.

Section 5: Building Your Own Patches

Once you have played with the patch for a while, you may become interested in creating a Pure Data patch yourself. In what follows we describe a short sequence of patches that we have created to illustrate how this is done. The first of these is `example1.pd` . Open this file and click on the object [modefilter0] to see how we have converted the `modefilter_standalone.pd` patch from above for use as a preliminary subpatch in `example1.pd` . First, you will see that all of the initialization that we had in `modefilter_standalone.pd` has been moved to the outer patch, including the object [window-gen]. Don't forget to reset the data directory if needed. Second, you will see that we have added inlets and outlets. The order in which these objects appear from left to right in the patch determine the order the inlets and outlets have in the outer patch. This is the reason you see [inlet] and [outlet] objects sometimes placed far away from what they are connected to. For inlets, we have chosen, from left to right, the following: a bang to start the processing, the day number, degree l, radial order n, azimuthal order m, width multiplication factor, and a toggle to turn playback on and off. For outlets, we have chosen, from left to right, the following: the output audio stream, the amplitude of the mode determined by the fit, and the name of the output array for this particular instance of [modefilter0]. By this time you have probably noticed that some of the connections between objects are drawn as thin lines while others are drawn bold. The difference is that thin lines carry control information, while the thick lines carry signal data, which is always processed at 44100 samples per second. Furthermore, it is conventional for objects that handle signal data to have names ending in '~'.

Note here a distinction in how Pure Data uses subpatches. Often, as you have seen here, the subpatch is loaded from a file of the same name with the `.pd` suffix. This type of subpatch is also called an abstraction. However, subpatches may also be defined as part of the parent patch, using the [pd] object. Here we have used this type of subpatch to hold arrays that needn't be visible on the front of the patch, or to make a patch more readable.

The inputs to [modefilter0] which must be given are the first five. In `example1.pd` we have left the width factor to take on its default value. We have connected a toggle to the final inlet, and we have connected the starting bang to this toggle so that everything will run with a single click. You may want to delete this last connection if you will be sonifying multiple modes, since the next time you click the bang it will turn off the toggle. The right two outlets are now connected for information only, but one might imagine using the amplitude to set the volume the mode is played at, for example. The amplitude units are arbitrary, but the values do accurately reflect the amplitude ratios between the modes as measured on the Sun. There are two parameters that should be the same for all instances of [modefilter0]: the playback sample rate and the frequency downshift factor. These two parameters are therefore set in the outer patch and broadcast using a [send] object (abbreviated to [s] in practice). Inside [modefilter0] the broadcast is received by the [receive] object (abbreviated [r]). To hear the output coming out of the left outlet, we must connect to the digital-to-analog converter, represented by the object [dac~], just as we previously did in `modefilter_standalone.pd` . (The object [audio_safety~] is one provided with this project; its purpose is to filter out corrupted data.)

You will also see that we have connected the output audio stream to the inlet of a [fiddle~] object. the documentation for this built-in object can be viewed by right clicking on it and selecting "Help". In short, it measures the pitch and amplitude of the stream on its inlet. Here, it tells us what tone we are actually generating at a particular playback sample rate and after shifting down in frequency.

Finally, the next step is to make a copy of the [modefilter0] object and everything connected to it, which you can do in Edit mode by highlighting the relevant boxes and selecting "Duplicate" from the edit menu. Move the new copy to wherever you would like to put it, and now you can listen to two modes at once, turning them on and off with the toggles connected to the right inlets. It works to have two copies of [audio-safety~] and [dac~], but common practice would be to have only one, and connect all the left outlets of the [modefilter0] objects to the same inlet of [audio-safety~]. It is one of the features of Pure Data that it automatically sums audio signals at inlets. Of course, one should also adjust the respective volumes of the modes to avoid clipping. You should end up with something like `example2.pd`, which has two [modefilter0] objects, but you may add as many as you like.

Aside from the second copy of [modefilter0], in `example2.pd` we have also added a calculation of the total transposition factor. This illustrates another important consideration to bear in mind, which is that a visual programming language does not explicitly specify the order in which operations are carried out. Furthermore, the default behavior for objects in Pure Data is for only their leftmost inlet to trigger the output. We call this the hot inlet and the other inlets cold. The canonical way to deal with this situation is with the [trigger] object (abbreviated [t]). As before, you can view its documentation by right clicking on it and selecting "Help". Basically this object distributes its inlet to its outlets in right to left order, converting as specified by its creations arguments. In the example shown in `example2.pd`, the downshift factor is sent first as a float to the cold inlet of the [/] (division) object, and then a bang is sent to the hot inlet of the [float] object (abbreviated [f]). Here, the built-in object [select] (abbreviated [sel]) is used to replace 0 with 1 and pass all other numbers unchanged. The [float] object serves to store numbers and output them when it receives a bang on its left inlet. The result is that the division will be performed regardless of the order in which the sample rate and downshift factor are specified. For more examples of using the [trigger] object, see `modefilter0.pd` . Note that the [float] object is often not needed because most of Pure Data's mathematical objects store the value of their left inlet automatically and reuse it when a bang is received. On the other hand, if space allows, explicitly using the [float] object can make code more readable.

(Note: if you open example1.pd and example2.pd at the same time, you will get error messages in the Pure Data console about the array window being multiply defined. Also, the first time you run any of the example*.pd patches, you will have to set the data directory and save, unless you are using the default data directory.)

Keep in mind that if you want to see what is happening inside the [modefilter0] subpatch, you can click on it and interact with it directly. You can view all the arrays involved, for instance, or change the value of the width factor. You can also still save the output array to a wav file as before. Each copy of [modefilter0] in your patch corresponds to a separate instance, and you can interact with each instance separately, although this can be confusing if you have many instances open at once.

If we want to have greater control over the synchronization of playback between various modes, we will need to use the output arrays generated by the various instances of [modefilter0]. As an example, we have created example3.pd, wherein we use the array names with the [tabread] object, which simply reads elements of an array. The other new object here is [until], which outputs a given number of bangs, here equal to the block size. (See the help for both these objects.)

Once the two output arrays have been created by the two instances of [modefilter0], you can simply click the new bang to create the sum of each multiplied by its amplitude, which will be displayed in the array sumtest. Click the message box with "sumtest normalize" to scale the new array so that its absolute value never exceeds one (necessary to avoid clipping). You may click the message box with "sumtest const 0" to reset this array to zero.

Perhaps you have noticed the absence of [trigger] objects in the new code. This was done to make the code more readable, but this practice should be avoided. Pure Data actually sends data along the connections in the same order that you made them in time, although this is invisible. As it happens, we made the connections in the correct order for the code to work, but there is no way for one to tell the order by looking at the patch.

We now move from "teaching" patches to real finalized patches. In example_addition.pd we have moved the mode addition logic into a new object [modeaddition] and correctly implemented the triggering. This new object takes as a creation argument the name of the array for the result. This array must be created on the parent patch first. We also added the ability to write the result out as wav file. To listen to the new array (be sure to normalize first!) we have put an [arbitrarySR] object on the parent patch, previously used only inside [modefilter0].

You will also notice that we have replaced [modefilter0] with a new finalized version of the base patch, [modefilter], which has subsumed the [fiddle~] object. We added two new outlets: one for the frequency read from the mode parameter file (in units of microhertz) and one for the frequency measured by the [fiddle~] object (in hertz). This allows us to use the calculated transposition factor to compare the input frequency (measured by fitting) to the output frequency (generated by the patch). [modefilter] also sends a bang to outdone once the resynthesis completes; this will be used in the next section.

Now let us suppose that we would like to add an arbitrary number of modes. One way to do so would be to modify [modeaddition] so that rather than take two arrays and two amplitudes as input, it would take only one of each and add the corresponding array to a running sum. We could execute this object once for each array we want to add and then normalize the result at the end. This is implemented in the object [modesum], illustrated in `example_sum.pd` . Because we are reading and writing from the same array, we need an additional [trigger] object to make sure this is done in the proper order. Further, we need to condition the array name somewhat, due to subtleties in the way Pure Data handles symbols (essentially repeating what is done inside [modefilter]).

Although there are too many overlapping connections to follow by eye, in `example_sum.pd` we now have five instances of the [modefilter] subpatch, and we have arranged them so that most of the inputs fall along the right edge of the screen (the toggles are especially enlarged for ease of use with a touchscreen). Once you have the modes you want loaded, you may add them up by clicking the corresponding bangs. The result is placed in the array sumhold, which is now placed in the subpatch [pd sumarray]. To listen to it, first normalize the array by clicking the corresponding message (top middle of the patch) and then play it with [arbitrarySR]. To create a new sum, first reset sumhold to zero. You may also save to file in either the [modesum] or [pd sumarray] subpatches, but notice the file name is a constant `modesum.wav`, so this file must be renamed if you want to save multiple sums.

Section 6: Extensions

Various properties of the Sun are known to change with an 11 year period; this variation is known as the solar cycle. Since we have 15 years of MDI data and 9 years of HMI data so far, we now have the opportunity to discover whether or not the effects of the solar cycle might be audible. In order to do so, we would like to concatenate our various output arrays together. For MDI, the available data span 76 72-day time intervals with 74 72-day timeseries (144 days of data are missing). As of this writing, HMI has been operating for almost 9 years, or 44 contiguous 72-day timeseries so far. This is a significant extension of our coverage of the solar cycle. Further, one might inquire as to whether there might be systematic differences between the two instruments during the time of their overlap.

An example of how one might concatenate timeseries is shown in `example_concat.pd` . Here you will also notice two new objects: [modecat] and [arbitrarySRmulti]. The first of these is quite simple: it takes the array named on its right inlet and copies it into the array named as a creation argument for the object, starting at the index given on its middle inlet. For this index to be calculated properly, you must first enter the day number of the first timeseries you will process. Inside [modecat] it is assumed that the target array is large enough, but we have already ensured this in the outer patch. As usual, you may write the resulting array to a file. To listen to it, the new object [arbitrarySRmulti] takes as a middle inlet the total number of timeseries that have been concatenated. The same window array will be applied to each segment. More complicated crossfading may be desired, especially in the case of MDI, which has one of its timeseries offset from the others. This is dealt with in the next example below.

We have also introduced the number of samples per day, sperday, which is 1440 for MDI and 1920 for HMI. In this connection, we also note that for a given sample rate, HMI will require a different downshift factor to achieve the same total transposition factor as we use for MDI, namely only 3/4 as much. Alternatively, if you want for the HMI timeseries to play in the same amount of time as the MDI timeseries, use a different sample rate and the same downshift factor.

Another change that we have made is that by default this patch will look for a different mode parameter file for each day number; this has both advantages and disadvantages. An advantage of using the averaged mode parameters is that the filtering of each timeseries will use exactly the same frequency interval. The peak frequency may shift within this interval, but the output sound would not be contaminated by the loss or addition of frequency bins at the edge of the interval. The disadvantage of using the averaged mode parameters is that we lose all information about how they change with time. On the other hand, using the variable mode parameters means that there is a chance we may run across an interval in which the fitting failed for a given mode. In any case, we now give the width multiplication factor a default value of 1.2 to somewhat mitigate the effect of varying widths and frequencies.

Of course if one is ambitious, then they could create new mode parameter files where some parameters are averaged and some are not. or, by interpolating mode parameters, one could attempt to use 36 day timeseries.

Finally, in `example_concat.pd` we have included an example of how to use the built-in object [qlist]. Its purpose is to act as a sequencer, sending messages to specified [receive] objects at certain times. It reads this information from a text file. We have included two such files, `qlist.mdi` and `qlist.mdi`, which list the available day numbers for the two instruments. For testing, you may wish to use a subset of one of these. For example, suppose we take the first 5 lines of `qlist.hmi` and place them in a new file, `qlist.hmitest` . To use it, we would then click the bang to set it as the sequence file. Then we would click the "hmi" message box and enter "6328" for the day number of the first timeseries. Then we can enter the desired mode's l, n, and m as usual, and click the bang to start the sequence. Before listening to the result, we would also enter "5" at the top of the patch for the number of timeseries.

The text file read by [qlist] contains lines of the form "time send_object message;". For exampe, the first line of `qlist.hmi` is "0 daynumber 6328;" which means at time zero send the message "6328" to daynumber. The next line is "14000 daynumber 6400;" which means to wait 14 seconds and then send the message "6400" to daynumber. The 14 seconds is the time it takes for the fft's to run: 262144/44100 ~ 6 seconds to run through the block, which must happen twice, and we add an additional 2 seconds for file loads and array processing. The remaining lines of `qlist.hmi` are of this same form. As each new timeseries is generated, the concatenation is triggered automatically when the resynthesis completes, with the [s outdone] inside [modefilter]. Once the sequence is finished running, you may listen to the result using [arbitrarySRmulti].

You may also use this patch to do the same thing with MDI. The file `qlist.mdi` contains the sequence for all 74 MDI timeseries, but for the number of timeseries you should enter "76". This will leave 144 days worth of the target array at zero, which is as it should be for missing data. As before, make sure that the "mdi" message box has been clicked and enter "1216" for the day number of the first timeseries. Of course you may edit `qlist.mdi` to make a shorter sequence; in that case adjust the day number of the first timeseries and the total number accordingly.

A close inspection of `qlist.mdi` will reveal the the timeseries beginning on day number 2116 is offset from the others by 36 days. The 108 days preceding and the 36 days following this timeseries are missing. Hence, if we use [arbitrarySRmulti] to listen to the full mission, the window array will not be applied properly to this one segment. Often this is not a problem, but it could be. Further, we have as yet no mechanism to apply any windowing to the files we write. If you listen to the concatenated timeseries without windowing, you will hear clicks between some of them.

These concerns are addressed in `example_concat2.pd`. We have added a new object, [applywindow], which simply multiplies the array specified on its right inlet with the window array and puts the result in the array named as a creation argument. We have also generalized the previous patch so that now one may add two modes together before concatenating, and each sum will be multiplied by the window beforehand. We now count the number of bangs sent to outdone and trigger [modeaddition] every two of them. [modeaddition] will trigger [applywindow] which will then trigger [modecat]. At the end of the sequence the final array will be in sumtest as before, which can be viewed by clicking [pd catarrary]. To listen to the result, you will need to manually normalize sumtest. Also, make sure the toggle connected to [s window-on] is set to zero, which it is when the patch loads. You may want to turn it on to listen to individual modes.

A somewhat more sophisticated implementation of the same idea can be found in `example_sequencer.pd`. As before, we have replaced [modeaddition] with [modecat], but we now need only a single instance of [modefilter]. Another notable change from the previous example is that we now use [qlist] in its other mode, where each line of the input file is retrieved with a "next" message. The first "next" message will immediately send every line that is not preceded by a number. Subsequent "next" messages will step through the rest of the lines. In our implementation the numbers at the beginning of these lines are unused, so we simply use "0".

We also now make use of the capability of [modefilter] to use an absolute number of frequency bins for downshifting, rather than downshifting by a multiplicative factor. This will no longer preserve the musical intervals between modes, but it will preserve the absolute frequency difference between them, making certain small differences audible.

We have connected new [receive] objects to specify the sample rate, downshift, and length of the output array. Hence, the first three lines of the qlist file could be

```
samprate 8000;
downshift 4;
numseries 10;
```

but one may still set these manually instead. The remaining global parameters, such as the instrument, whether or not to use averaged mode parameters, and the length of the ramp used to generate the window, can also be set in this way if values other than the default are desired.

The remaining lines of the qlist file will take one of two forms. First, one provides a line for every mode they wish to add together. These lines define a combination of day number, l, n, m and width multiplication factor. The whole list is sent as daylnm, which is then parsed by the new object [parsedaylnm]. For example, one such line could be

```
0 daylnm day 1216 l 1 n 18 m 1 width 10;
```

which means to send the entire message "day 1216 l 1 n 18 m 1 width 10" to daylnm. The elements of this message will be picked out two at a time and used to send the 5 inputs needed. The first such line must specify at least the 4 required numbers, but subsequent lines need only specify changing values, just as if you were using [modefilter] directly. The order is unimportant. Of course you may also manually enter any values that you wish to remain constant before starting the sequence. When [parsedaylnm] gets to the end of the message it received, it sends a bang to startbang.

Next comes a single line telling where to put this combination of modes in the output array. For instance, the first such line would typically be "0 dayindex 0;". Then after a number of daylnm lines, the second would typically be "0 dayindex 72;" and so on. However, one may specify whatever positions they want, for example leaving silence in between the segments, or even partially overwriting previously written segments.

When the sequence is finished, the [qlist] object sends a bang to normalize the output array. You may listen to it using [arbitrarySRmulti] as before. You may also write it to file inside the [pd catarray] subpatch.

UPDATE: you may now use timeseries of length other than 72 days. Such timeseries may be generated using a function provided with the SoshPy Visualization Package; any length that is an even divisor of 72 is permitted. To use such timeseries, click the corresponding message connected to [s ndays]. For now this will only work with averaged mode parameters.

Section 7: Conclusion

We hope that the level of detail presented here has allowed you to effectively use the SoSH tool, and perhaps even given you some ideas about new directions the project could take. Contributions are welcome from everyone, even if it is only the idea; do not hesitate to contact us if you need help implementing it or simply think it would be a valuable addition to the tool. New applications are likely to be added to the project as time goes on, so check back with us if you want the most recent version. If you have discovered combinations of solar tones that you find pleasing, either aesthetically or scientifically, feel free to share them with us, especially if you would like them to appear in our online gallery.

For further reading, we refer you to the documentation for the SoshPy Visualization Package. It contains an extended discussion of helioseismology as well as a description of software you can use to create graphical representations of solar oscillation modes.

Contact the developers tim larson at tplarson@sun.stanford.edu or Seth Shafer at sethshafer.com. The website for the Sonification of Solar Harmonics Project is http://solar-center.stanford.edu/SoSH/ .