Visualizing the Sun:

An Introduction to the SoshPy Visualization Package

This package was created as part of the Sonification of Solar Harmonics (SoSH) Project. An extensive discussion of helioseismology with audio samples can be found at http://solar-center.stanford.edu/SoSH/ . The following is intended to augment the discussion in the documentation provided with the SoSH tool (`instructions_audio.pdf`). Installation instructions can be found in `README_visual.txt` .

Contents

Section 1: Helioseismology

The study of oscillations inside the Sun is called helioseismology. In particular, we are here concerned with acoustic waves for which the Sun provides a resonant cavity. The Sonification of Solar Harmonics (SoSH) Project is a collaborative research initiative that seeks to transform helioseismology into a listening experience for the scientist and nonscientist alike. It has developed the SoSH Tool, which is able to sonify solar acoustic data by filtering and transposing it up to the range of human hearing. The SoshPy Visualization Package was created to provide complementary graphical representations of the solar harmonics being sonified.

Consider that we are here concerned with sound, which is simply the coherent vibration of matter. Turbulent convection near the solar surface excites sound waves, and those with frequencies that resonate form the harmonics, also known as normal modes of oscillation. Now imagine a plasma element (a chunk of matter) at the surface of the Sun. In response to a given mode, which inhabits the entire Sun, this single plasma element will oscillate at the mode frequency. In other words, it moves back and forth; some part of this motion will be in the radial direction, some in the latitudinal direction, and some in the longitudinal direction. A model can tell us how much of the motion lies in each direction. We call the radial motion the vertical displacement, and we put the latitudinal and longitudinal components of the motion together to make the horizontal displacement.

Now, how are we to measure this displacement? Actually, it turns out to be much easier to measure the speed of our plasma element, which is simply the rate at which the displacement is changing. In mathematical terms, we say the velocity of the plasma element is the derivative of its position. Since the plasma element is moving in response to one of the Sun's harmonics, it can be modeled as a simple harmonic oscillator. In other words, its motion is mathematically equivalent to the motion of a weight attached to a spring. In particular, the model tells us that the magnitude of the the plasma element's velocity is equal to the magnitude of its displacement multiplied by its frequency, which is simply the mode frequency.

It is the velocity of the plasma element that we are able to measure. The details of this measurement are beyond the scope of this discussion, but in short we are actually measuring the frequency of spectral lines of atoms at the solar surface. For plasma at rest, these spectral lines have precise frequencies that can be measured in a laboratory. The motion of the plasma causes these frequencies to be shifted slightly. This is called the Doppler effect, and it is exactly the same phenomenon that you can observe with sound waves when a siren drives past: when the siren is approaching it sounds higher, and when it is moving away it sounds lower. We use the Doppler effect to create velocity images of the Sun's surface, and for this reason we call them dopplergrams. Each pixel of the image corresponds to one of our plasma elements, and the pixel value tells us how fast that element is moving toward or away from the observer.

Here we run into a fundamental limitation of our method of observation. We would like to know both the vertical and horizontal components of the velocity. Unfortunately, a dopplergram only gives us the velocity along the observer's line of sight. At the center of the solar image, this is exactly the vertical component, and at the very edge, it is the horizontal component. Everywhere else we measure a mix of the two, and nowhere can we measure both of them. This is one of the reasons we have to rely on a model.

So what quantity should we plot to represent a mode? It turns out that for a large majority of modes, the horizontal component is very small compared to the vertical component at the surface, and for many purposes can be neglected entirely. Hence, by default we will show the vertical component of velocity only. The scripts described below, however, are also able to plot the latitudinal and longitudinal components, as well as these two combined into the total horizontal component. They can also plot the magnitude of the total velocity, as well this magnitude squared.

We are also able to plot either a surface view or an interior view. In the latter case, we will show a plane containing the solar rotation axis. Also in this case we plot the velocity scaled by the square root of the background density. This is for two reasons. Firstly, since the velocities drop off rapidly with depth while the density increases rapidly, the quantity plotted shows visible variations throughout the interior. Secondly, the square of this scaled velocity is actually equal to the energy density of the mode.

Note that for every quantity plotted, the colormap will be scaled to use the full range of the data. Therefore, in a surface view, the velocity squared is indistinguishable from the energy density, because the background mass density is constant over the solar surface. Similarly, for plots of single modes, we do not scale the displacement amplitudes by frequency to get a velocity, because this would also be invisible. When we begin adding modes together, however, we must properly account for their relative amplitudes, so in this case the frequency scaling is necessary because the frequency will be different for different modes.

Section 2: Surface Views - `drawharmonics.py` and `addharmonics.py`

For instructions on running these scripts, see appendix A. In what follows i give a more detailed account of how they work. We start with drawharmonics.py, which plots surface views of single modes.

But first we need a small amount of mathematical background. An oscillation mode on the Sun, also known as a harmonic, is given by the product of a function of latitude and longitude and another function of radius only. The first function is called a spherical harmonic, and for these we have explicit analytical expressions. The function of radius is called a radial eigenfunction, and these must be computed numerically by solving a system of differential equations.

For a surface view, we only need the spherical harmonic, which is given by

$Y_{lm} = P_{lm}(\cos\theta)*\exp(im\phi)$

where the P_{lm}'s are associated Legendre polynomials. In the coordinate system that we use, θ (theta) is the angle from the solar rotation axis, so it is different from the latitude by 90 degrees. The angle ϕ (phi) is exactly the longitude. Each spherical harmonic is described by two integers, l and m. The spherical harmonic degree l is equal to or greater than zero 0, and the azimuthal order m ranges from $-l$ to l. The associated Legendre polynomial, however, depends only on l and $|m|$, the absolute value of m. More details can be found in **`instructions_audio.pdf`** .

At the beginning of the script, the arrays `theta` and `phi` are generated by the function `image2sphere()`. This function is exactly the same as that used to generate artificial spherical harmonics for scientific analysis. It takes as arguments the number of pixels in the x and y directions (default 1000) as well as three parameters describing the orientation of the solar image. The first of these is `bangle`, which is defined as the latitude of the subobserver point. In other words, it is the tilt of the solar rotation axis towards the observer. Similarly, the parameter `pangle` is the tilt in the plane of the image. These angles default to 30 degrees and 0 respectively. A third parameter, `distobs`, gives the distance of the observer in solar radii. It defaults to 220 (the actual distance between the Sun and the Earth is about 215 solar radii), but this can be set lower or higher to zoom in or out of the solar image. One may also specify `xoffset` and `yoffset`, which are pixel offsets in the x and y directions.

The script then enters an input loop wherein the user is prompted to enter the l, m, and n of the desired mode. This final integer, the radial order n, describes the behavior of the mode along the solar radius. For a surface view, it serves only to determine the ratio of the vertical and horizontal components.

Next we must calculate the associated Legendre polynomial. This is ultimately done by the function `setplm()`, which again is exactly the same routine used for scientific analysis. To find the P_{lm} for a given l and m, we start by using an explicit expression for the P_{lm} with $l=m$, and then use a recursion in l to find the P_{lm} with the requested l. Hence, a total of l-m P_{lm}'s are actually calculated, along with their corresponding derivatives. To save on computing, these are all saved for potential use in subsequent iterations of the loop. Therefore, if you plan to plot several modes with the same m, you should start with the one with the highest l. The list of P_{lm}'s already calculated is saved in the module, so if you are running in ipython, it will be preserved in between invocations of the script

Next we construction the spherical harmonic, Y_{lm}, as well as its derivative in both the θ and ϕ directions. The derivative in the θ direction gives the θ component directly, and dividing the derivative in the ϕ direction by sin(θ) gives the ϕ component. The radial component is given directly by the Y_{lm}. In all cases, we must take the real part to find the physically meaningful quantity.

Next, we read the vertical to horizontal ratio from a table. This table was extracted from a model using the function `writesurfacemodel()`. The table contains both the magnitudes of the vertical and horizontal components of the displacement at the solar surface as well as a theoretical estimate of their ratio. More details of the model are provided in the next section. The script **`drawharmonics.py`** arbitrarily gives the vertical component (`rc`) a magnitude of 100 and sets the magnitude of the horizontal component (`hc`) according to the theoretical ratio.

Finally, the python dictionary `varlist` is populated with the six plotting options:
`Vr` - radial or vertical component
`Vt` - theta or latitudinal component
`Vp` - phi or longitudinal component
`Vh` - magnitude of horizontal component, `=sqrt(Vt^2+Vp^2)`
`Vmag` - magnitude of total velocity, `=sqrt(Vsq)`
`Vsq` - magnitude squared, `=(rc*Vr)^2+(hc*Vh)^2`
where i have used the shorthand `Vr` for `varlist['Vr']` and so on. Note that `Vr`, `Vt` and `Vp` are signed quantities, while `Vh`, `Vmag` and `Vsq` are strictly positive. Once the script terminates, all six quantities will be available in `varlist`.

Unless the script was started with `animate=0`, the plot will be animated. To do so, it uses the two functions defined at the top of the script. The first, `calcylmt(t)`, serves to evolve the mode in time. To do so, it require the arrays `ylm`, `dylmt` and `dylmp` to already exist. The time dependence is given by exp($-i\omega t$), but rather than using the mode's actual frequency, we simply set ω=2π/`nframes` so that the animation will loop seamlessly. `nframes` is the total number of frames in the animation and can be set when the script is called. Note that `calcylmt(t)` only calculates the requested quantity for each frame, rather than the full six that were calculated for the first frame.

The second function needed for the animation, `ylmanimate(t)`, simply takes the array returned by `calcylmt(t)` and uses it to set the data for a previously existing image instance. This is the function that must be passed to the matplotlib animation utility, and it is also called to save frames to disk.

We now move on to the script **`addharmonics.py`**, which as the name suggests plots sums of modes. It is a relatively straightforward extension of the previous script, but the first notable difference is that the input loop does not accept the list of modes to plot. Rather, this list is set once at the beginning of the script. That is, the user first enters the number of modes, followed by the l, m and n of the desired modes. The script then enters its input loop, wherein plotting parameters may be changed. In other words, to plot a new combination of modes, the script must be rerun.

For each mode, the Y_{lm} and its derivatives are calculated as before and the three resulting components are stored in lists. The same table is read to get the model values, but now the vertical (rc) and horizontal (hc) magnitudes are used directly. We will also scale the displacement amplitudes by frequency (freq) to get the proper relative velocity amplitudes. These three parameters are likewise stored in lists.

Next the sums are performed and the six plotting variables are stored in varlist as before. Now, however, when we animate the plot, we give each mode in the sum its proper relative frequency. In particular, we use $\omega=2\pi$*freq*freqscale/nframes, where freqscale is a parameter that can be set when the script is called. Because the frequencies have already been scaled to be in units of millihertz, the default value of freqscale is 1/3. In general, the animation will no longer loop seamlessly. To see the sum evolve further in time, one may increase freqscale. One may also wish to simultaneously increase nframes. Note that the animation functions in this script are named calcsumt(t) and sumanimate(t).

Here we must insert a short discussion of colormaps. For the signed quantities Vr, Vt and Vp, we would typically like for the value of zero to map to the center of the colormap. For a single mode, this is almost always accomplished automatically, because the data range will usually be symmetric around zero. Depending upon the orientation of the solar image, however, certain modes (often those with m=0) will not satisfy this requirement, perhaps leading to unexpected results. To correct this situation, by default we force the color mapping to be symmetric around zero for the signed quantities. This means that the full range of the colormap may not be used. To revert to automatic scaling, one may specify colorshift=0 when starting the script. The default value is 1. In either case, the color mapping is determined by the data in the first frame. This is adequate for single modes. For sums, however, it may happen that a subsequent frame will exceed the data range of the first frame, in which case the colormap will saturate. This is usually not a problem, but if one wishes to avoid this behavior, one may specify colorshift=2. In this case, the script will step through calcsumt(t) for t in range(nframes) and find the full range of possible data values in advance. We may also add the capability to change the range as the animation proceeds, but this is not yet fully implemented.

Section 3: Interior Views - `drawradial.py` and `addradial.py`

These two scripts operate in close analogy to their surface counterparts. The most important difference is that these require a much more extensive model for input. In particular, the full radial eigenfunctions must be read from a file, along with the mesh on which they were calculated. The full eigenfunction actually consists of two functions, one giving the amplitude of the displacement in the vertical direction, and one giving the amplitude of the displacement in the horizontal direction. We call these two functions ξ_r and ξ_h respectively (xir and xih in the code), and they each depend on both spherical harmonic degree l and radial order n. Finally, we will also need the background mass density used to construct the model. As mentioned above, this will be used to scale the radial eigenfunctions.

All of this data has been packaged into an hdf file available from the SoSH website: http://solar-center.stanford.edu/SoSH/#mods . At the beginning of each of these scripts, we call the function `loadmodel()`, which reads the mesh and mass density used for the model, as well as the l, n and frequency of all the modes in the model, and puts them in global variables. A second function, `getradial(l, n)`, will return the actual displacement eigenfunctions for the modes we wish to plot.

After `loadmodel()`, the scripts call `image2rtheta()`, which returns the arrays containing the r, θ and ϕ coordinates for each point in the image (`r`, `theta` and `phi` in the code). Like its surface counterpart, this function can take as arguments the number of pixels in the x and y directions (default 1000). It also accepts `distobs`, the observer distance in solar radii, which defaults to 220 as it does for surface views. You may also specify `xoffset`, `yoffset` and `pangle` (all default 0), which also have the same meanings as above. In general, these parameters should be set the same for interior views as for surface views if you intend to compare them. `image2rtheta()` can take two more parameters: `bangle` and `gamma`. Here `bangle` has a slightly different meaning; the data are foreshortened in the vertical direction by simply dividing by cos(`bangle`) before rotation by `pangle`. In the same way, `gamma` provides the foreshortening in the horizontal direction. This foreshortening may be useful if you want to overlay the interior views on top of a surface view.

These scripts can also take another parameter, `rsurf`, which is the fractional image radius at which to truncate the model (default 1.0). In some cases you may wish to remove the surface values if you find they dominate the plots. This will reduce the size of the area occupied by the plot.

The value of ϕ returned by `image2rtheta()` will be 0 on the right half of the image and π (180°) on the left. Then in **`drawradial.py`**, we have chosen to add $\pi/4m$ to `phi`, so that the real and imaginary parts of the spherical harmonic (recall the factor of exp($im\phi$)) will be the same. This means that all three velocity components will have comparable magnitudes in the first frame. In **`addradial.py`**, we set ϕ according to the first mode entered. In the case of m=0, the imaginary part is identically zero and we add nothing.

Appendix A: Usage Instructions

Here we provide usage instructions and a full description of parameters for scripts in the SoshPy Visualization Package, which was written to provide graphics to accompany the Sonification of Solar Harmonics (SoSH) Tool. Refer to **`README_visual.txt`** to ensure you have the needed modules installed.

A1: **`drawharmonics.py`**

The simplest of these scripts is **`drawharmonics.py`** , and in its simplest invocation it can be run either at the command line with only "python drawharmonics.py" or in an ipython session with "run drawharmonics". In both cases you will then be prompted to enter a single mode's spherical harmonic degree (l), its azimuthal order (m), and its radial order (n), although this last integer will only be used when displaying certain variables. By default, the script will plot the radial component of the mode's velocity, in which case n is unused. After the mode is calculated, the script will display an animation of the mode on the solar surface. Once you close the plot window, you will be prompted to enter another mode's l, m, and n. To quit the script, enter 'q' at any time.

Of course the detailed behavior of the script can be changed with a variety of options, some of which can be specified when the script is called, and some of which can be specified interactively. To use the latter functionality, you may enter 'c' at any time to change a plotting parameter. You will first be given the opportunity to change the colormap. The default is 'seismic'. You may enter 'l' to print a list of preinstalled colormaps, along with the colormap currently in use. You may use your own colormap if you have registered it with `cm.register_colormap()`. Simply hit enter to retain the current colormap.

You will then be prompted to enter a plotting variable. Again, you may enter 'l' to list options along with the variable currently in use. The options are the six found below:

`Vr` - radial or vertical component of velocity.
`Vt` - latitudinal or theta component.
`Vp` - longitudinal or phi component.
`Vh` - horizontal component, `=sqrt(Vt^2+Vp^2)`.

So far, we have not needed the value of n. For surface plots such as these, the value of n only serves to define the ratio of the vertical component to the horizontal components of the velocity. Hence, n is only used to determine the total velocity:

`Vmag` - magnitude of the total velocity `=sqrt(Vsq)`.
`Vsq` - square of the total velocity, `=(rc*Vr)^2+(hc*Vh)^2`. This is proportional to the energy density.

These six strings are the indices to python dictionary `varlist`, so properly speaking, i have employed the short hand `Vr=varlist['Vr']` and so forth. Note that `Vr`, `Vt`, and `Vp` are signed values whereas `Vh`, `Vmag`, and `Vsq` are unsigned. Also, for the large majority of modes, the vertical component at the surface will be much larger than the horizontal component.

Finally, you have the option to save every frame of the animation as a png file. Entering anything other than 'y' turns saving off. These files will be written into a subdirectory of the "png_out" directory. You may subsequently use ffmpeg to render them as a single video file, such as an mp4 or gif.

You will now be returned to the query for l, m, and n. If you simply hit enter, the value from the previous iteration will be used. Hence, if you simply want to see what the previous mode looks like with the new colormap for instance, just hit enter three times. Or, for example, to save the mode you are currently looking at, close the plot window, enter 'c' and follow the prompts to turn on saving, then hit enter three times.

Many other parameters may be specified when the script is called. These are the following:

`pixels` - specifies the resolution of the animation by giving the number of pixels in both the x and y directions, so the total number of pixels will be `pixels`^2. default is 1000.

`distobs` - observer distance in solar radii, default 220. This value can be changed to zoom in and out of the image.

`bangle` - tilt of the solar rotation axis towards the observer in degrees. default is 30.

pangle - tilt of the solar rotation axis in the plane of the image in degrees. default is 0.

xoffset - pixel offset in x direction, default 0.

yoffset - pixel offset in y direction, default 0.

colorshift - set to zero to use colormaps with default scaling. For the signed quantities, this could result in the value of zero not falling in the center of the colormap, giving unexpected results. Set colorshift=1 (the default) to adjust the scaling for those quantities. Set colorshift=2 to step through all frames of the animation and set the range accordingly.

dpi - sets the resolution of the output images in "dots per inch". Default is 300.

nframes - number of frames in the animation, defaults to 64.

figsize - size of plotting window, defaults to 5.

animate - set to zero to plot a still image instead of an animation. If saving is turned on, only a single png will be written. Default is 1.

show - set to zero to turn off image display. Typically useful only if you are writing images to disk.

In an ipython session, the colormap and plotting variable will be saved between invocations of the script, as will the most recent values of l, m, and n. Saving, however, must be turned on with every invocation.

A2: `addharmonics.py`

As the name suggests, this script can plot sums of harmonics. Hence, the first thing you will be prompted to input is the number of modes. Bear in mind that the more you specify the slower the script will run. You will then be prompted to enter the l, m, and n of the modes you wish to add. You may reuse the previous value by simply hitting enter. In other words, if you wish to plot 3 modes with the same value of l or m or n, you only have to enter it once. For modes with the same m, the script will run faster if you start with the one with highest l.

As before, you may enter 'q' at any time to quit or 'c' to change plotting parameters. Once you close the plot window, you will enter a loop allowing you to change plotting parameters or save frames to disk. That is, to enter a new combination of modes you will have to run the script again. However, to plot the same set of modes the next time you run, you may specify a negative number for the number of modes. This will cause the script to look backwards that number of entries, rather than the default of 1. In this case, you may skip through the entry of l, m, and n by hitting enter an appropriate number of times. Or, for example, you may leave all modes the same but one.

This script also has an additional input parameter, freqscale. This is the floating point value by which to multiply the frequencies of the modes. These frequencies come from a model. Since they are given in units of millihertz and the Sun's peak acoustic power occurs at about 3 mHz, the default value of freqscale is 1/3 . To see the sum evolve further in time, increase freqscale; you may want to simultaneously increase `nframes`. Both `freqscale` and `nframes` can be changed interactively in the input loop.

When we evolve the sum in time for the animation, it may so happen that a frame will exceed the data range of the first frame, which is what we use by default to set the color scaling. By setting `colorshift`=2, you can tell the script to scan through the frames ahead of time and set the range of the color scale accordingly. However, the default setting would only result in some areas of the image being saturated, and this may be perfectly acceptable for many purposes.

Another difference is that the value of n is used to compute all plotting variables, because the modes are added together according to their relative surface magnitudes, which are given by a model. Also note that because we are plotting velocity, the amplitude of each mode is the amplitude of the displacement eigenfunction scaled by the frequency.

Finally, note that if you save the output, the files will be labelled only be the final mode you entered.

A3: `drawradial.py` and `addradial.py`

This script and the next plot interior views of the Sun. Specifically, they plot the amplitudes of modes on the radius-theta plane, which is to say a plane containing the line of the Sun's rotation axis. To do so, they must read the radial eigenfunctions from a model, which has been computed by numerically solving a system of differential equations. This model can be downloaded from http://solar-center.stanford.edu/SoSH/#mods and you should unpack it in your "sosh" directory.

These scripts no longer plot the modal velocity, but rather the velocity scaled by the square root of the background density. This is for two reasons. Firstly, since the velocities drop off rapidly with depth while the density increases rapidly, the quantity plotted shows visible variations throughout the interior. Secondly, the square of this scaled velocity is actually equal to the energy density of the mode.

Finally, these scripts can take a few extra input parameters in addition to those accepted by the scripts above. These are

`rsurf` - fractional radius at which to truncate the display, default value 1.0. You may wish to specify a lower value if you find a plot dominated by the amplitude at the surface.

`bangle` - almost the same as the parameter above, but here it simply foreshortens the data in the vertical direction by dividing by cos(`bangle`) before the rotation by `pangle`. Default is 0.

`gamma` - provides foreshortening in the horizontal direction, default 0.

As above, `addradial.py` accepts `freqscale`, while `drawradial.py` does not.

A4: `sosh.py` reference

Following is a list of all functions in `sosh.py`. There are explained in the full documentation.

loadmodel() - loads radial mesh, density, frequency, l, and n into rmesh, rho, modelnu, modell, and modeln.
getradial(l, n) - returns vertical and horizontal eigenfunctions as a tuple.
writesurfacemodel(lmin=0, lmax=300, rsurf=1.0) - writes file `model.surface.modes` containing l, n, frequency, velocity amplitude estimate, displacement ratio, surface vertical displacement, surface horizontal displacement, inner turning point of the mode in fractional radius, and mode energy for all modes between lmin and lmax at a fractional radius of rsurf.
image2sphere(xpixels=1000, ypixels=1000, distobs=220.0, bangle=30.0, pangle=0.0, xoffset=0.0, yoffset=0.0) - returns arrays phi and theta giving the spherical coordinates corresponding to pixel coordinates. See text for explanation of parameters.
image2rtheta(xpixels=1000, ypixels=1000, distobs=220.0, xoffset=0.0, yoffset=0.0, bangle=0.0, pangle=0.0, gamma=0.0) - returns arrays r, theta, and phi giving the spherical coordinates corresponding to pixel coordinates. See text for explanation of parameters.
setplm(lmax, m, x, plm, dplm) - calculates associated legendre polynomials for l=m to l=lmax at coordinates x (=cos(theta)), storing the result in plm and its derivative in dplm. Returns only the last such polynomial calculated.
querylmn(index) - prompts user to enter l, m, and n. Appends the results onto lists lsave, msave, and nsave before returning it. If index is negative, looks backward this amount in lsave, msave or nsave for default value. Otherwise the default will be the most recent value entered. If 'q' is entered returns (-1,-1,-1).
queryplotparms() - queries the user for three parameters: the colormap to use, the variable to plot, and whether or not to write to png files. If 'q' is entered, it is returned.
catchc(prompt, saveval) - prompt user with string prompt. If 'c' is entered call queryplotparms(). If 'q' is entered, it is returned. If nothing is entered, return saveval as a string.
printcmaps() - prints a list of colormaps that ships with matplotlib.
drawfigure(var, fsize=5) - creates a square figure of size fsize and plots the array var within it. Returns the figure and image instances as a tuple.
savefigure(ianimate=1, label='') - If ianimate=0, writes a single png file, using label in the file name. Otherwise, creates a new directory using label in the name, and writes nframes png files with the frame numbers as names.

Appendix B: SoSH Tool Utility Functions

This appendix provides instructions for using the soshdata python module to retrieve solar data for use with the Sonification of Solar Harmonics (SoSH) Tool. Refer to `README_visual.txt` to ensure you have the needed modules installed.

B1: data retrieval

Two types of data are needed. Ascii tables containing mode parameters and wav files containing the raw acoustic data. To retrieve these, first start an ipython session. Then enter these three lines:

```
import soshdata
soshdata.getmodeparms()
soshdata.getwavfiles()
```

After each function call you will be prompted to enter an instrument and a day number. The two instruments available are the Michelson Doppler Imager (MDI) and the Helioseismic and Magnetic Imager (HMI). When prompted, enter either "MDI" or "HMI" (case insensitive).

For day number, enter one from the list found in `daynumbers.txt` . MDI data spans day numbers 1216-6616. HMI data begins on day number 6328 and is ongoing, currently available through day number 9424. In the case of mode parameters, you may enter "average" instead of a day number, in which case an averaged mode parameter file is downloaded. In the SoSH Tool, you may elect to use such a file for all day numbers.

For the wav files, you will additionally be prompted to enter a spherical harmonic degree (l) and azimuthal order (m). For degree enter a number l=0-300. For order enter a number m=0-l. You may enter a negative value for m, but the resulting files will be the same, since they contain both signs of m already.

The native format of the data is the flexible image transport system (fits) and all m's for a given l are stored in the same file. The `getwavefiles()` function will retrieve this file, and then pick the m you have requested and write it out as two wav files, one each for its real and imaginary parts (m=0 has only a real part). If you subsequently request the same l and a different m, the fits file will already exist and not need to be downloaded. If you prefer for fits files to be deleted every time, instead run `soshdata.getwavfiles(delfits=True)`.

Instead of waiting for the functions to prompt you, you may specify their parameters. For example,

`soshdata.getwavfile(instrument='mdi',daynumber=1216,l=10,m=5)` or
`soshdata.getmodeparms(instrument='hmi',daynumber='average',lmin=50,lmax=100)`

The last two parameters shown for `getmodeparms()` allows you to save some disk space by limiting how many degrees are written into the "msplit" file. This file is not strictly required, however, and its creation can be disabled altogther by specifying `makemsplit=False`. For a full explanation of the use of the msplit file, see **`instructions_audio.pdf`** that came with the SoSH tool.

The order of parameters specified with "=" is unimportant.

The package includes a separate file to automate the use of these two functions with the SoSH tool, **`getdata.py`**. It runs in a loop taking data from from its standard input, which should be connected to a "pdrecieve 1991" on the command line (this single command has also been encapsulated in the script **`getdata`**). Inside the SoSH tool the "netswitch" toggle must be clicked, and then whatever data is needed will be retrieved automatically. When it is done, **`getdata.py`** uses pdsend to send confirmation back through port 9119. The pdreceive and psend utilities are included with Pure Data.

B2: data manipulation

The function splitwavfile() allows one to divide a wav file into segments that divide the input evenly. If no wav file is given, the user is queried for the instrument, day number, number of days, spherical harmonic degree l, and azimuthal order m. By default, the wav files containing the real and imaginary parts of the spherical harmonic are both split. As an example, the default length of 72 days (the length of all timeseries downloaded) can be split into lengths of 2, 3, 4, 6, 12, 18, 24 or 36 days.

The function scanspectrum() is able to reproduce some of the functionality of the SoSH tool. It takes a wav file as input or queries the user if none is given. It then searches for the wav file with the other component (real or imaginary) and then performs a fourier transform of the resulting complex array. Based on its other input parameters, it takes an inverse fourier transform of a window moving in frequency. These parameters are firstbin, lastbin, nbins, ndt, binstep and downshift. It starts at the frequency bin given by firstbin and takes the following nbins frequency bins. For each bin, it generates a sine wave shifted in frequency by downshift for ndt time steps. It then shifts forward in frequency by binstep (defaults to nbins/2) and repeats the process until lastbin is reached. That is, by default, the frequency windows will overlap by a factor of 2. If the input wav file contains N samples, the number of frequency bins will nf=N/2. The first half of the fft corresponds to positive m and the last half corresponds to negative m. The default value for firstbin is nf/6 and for lastbin it is 0.9*nf. A final parameter, ramp, gives the number of milliseconds over which to ramp up at the beginning of each segment and ramp down at the end. The default is 50.

The resulting array will be written to the file `output.wav`, and is also returned. To mimic the output of the SoSH tool, read the parameters "center bin" and "width in bins" that it generates. Then set firstbin=centerbin-width, lastbin=centerbin+width, and nbins=lastbin-firstbin. However, scanning the entire spectrum will allow you to hear the relative amplitudes of all the frequency components in the data.

B3: `soshdata.py` reference

Following is a list of all functions in `soshdata.py`.

getmodeparms(instrument=None, day=None, lmin=0, lmax=300, makemsplit=True) - queries user for instrument and day if not given. It calls getmfile() and then if makemsplit=True and the "msplit" file does not exist, it calls mkmsplit(). Note that day may also be the string 'average'. It returns the names of requested files.

getmfile(inst, daystr) - retrieves mode parameter file from the internet, unless it already exists, and returns its name.

mkmsplit(mfile, lmin=0, lmax=300) - uses mode parameter file to generate "msplit" file and returns its name.

getwavfiles(instrument=None, day=None, l=None, m=None, delfits=False) - queries user for instrument, day, spherical harmonic degree l and azimuthal order m if not given. If the wav files do not already exist it then calls getfilesfile() and converttofits(). If delfits=True, it deletes the downloaded fits file. It returns the names of requested files.

getfitsfile(inst, day, l) - retrieves the needed fits file from the internet, unless it already exists, and returns its name.

convertfitstowav(ffile, hind, m, wfile) - reads the selected slice from the fits file and writes the real part out as as wfile. It writes the imaginary part into a file with the same stem. hind is the index of the fits hdu, which is 0 for MDI and 1 for HMI. It returns the names of the generated files.

apols(lin, amaxin) - generates the polynomials needed to expand the a-coefficients, called by mkmsplit().

splitwavfile(wavfile=None, splitfactor=2, splitboth=True) - divides input into splitfactor segments.

scanspectrum(wavfile=None, downshift=4, firstbin=None, lastbin=None, nbins=200, ndt=10000, binstep=None, ramp=50) - scans the fourier transform of the input and returns the inverse fft of a moving frequency window. See documentation above for details.