Near Real-Time EMD Filter Without Distortion

A low/band-pass filter for when your signal is very low frequency.

Abstract

Filters are great, except that they will always add defects or artefacts like group delay, or always take some time to relax or decay, and if you have signal near the cut-off frequency or a really low frequency filter, there can be significant distortion, modifying your signal such that it's no longer useful (i.e. as input to a real-time estimation scheme). Like the Fourier Transform, Empirical Mode Decomposition (EMD) is a way of breaking up a signal into its constituent parts, but instead of sines or cosines, EMD gives you Intrinsic Mode Functions (IMFs). By looking at the instantaneous frequency of each IMF, it's possible to get rid of the ones that are obviously noise, reconstructing a good approximation of your noiseless signal. While more computationally intensive than standard DSP techniques (more-so than even FIR filters), you no longer have to worry about group delay, exponential decay due to time constants, or other filter defects that occur with real-time filtering.

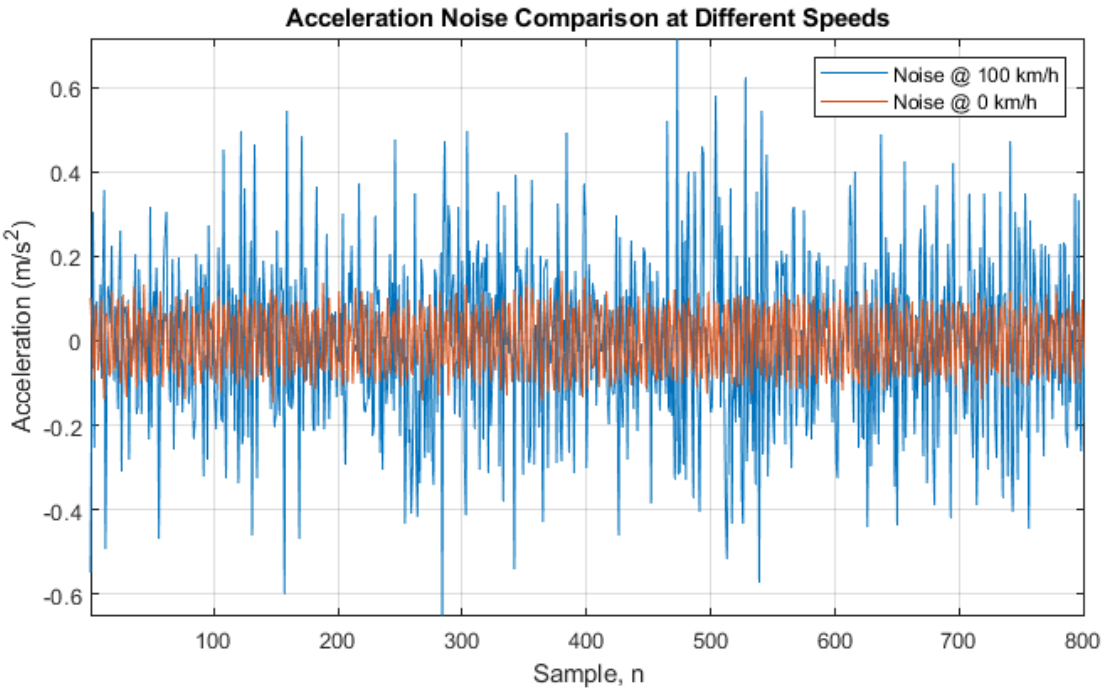
You know, back when I was a laser scientist, group velocity dispersion... was a thing. My work was primarily concerned with the inter/intra-modal group velocity dispersion (different modes in multimode fiber travel at different speeds), and it was a really big deal. However, even frequency-dependent group velocity dispersion (different frequencies of light travel at different speeds) can become quite significant over medium distances (10s of kilometers) at high rates of on-off keying (the bandwidth of a 10 Gbps square signal is... 10 GHz). It's such a significant defect that much work has been put into correcting for it, with fancy chirping signals or even short lengths of fiber with the opposite, but much higher, frequency-dependent chromatic dispersion.

So then why in the hell when we talk about digital signal processing is it barely even mentioned? I guess mostly because it's not a big deal; linear-phase FIR filters have a linear phase response so group delay is a constant across all frequencies of interest, so things just get pushed back in time a bit. However, what if you need to use the output of the filter more or less in real time as input to another system? Also, what if the noise you're trying to filter out is non-stationary?

In my current work, that's exactly what we have to deal with: we need to get rid of noise from accelerometer readings so that they can be useful in real time. But here's the problem with the accelerometer data that I'm trying to work with:

1. It's very low frequency (less than 2 Hz, mostly less than 1 Hz),
2. Short-term bias in the noise tends to compound to extreme error when you want to use it to measure accumulated quantities like velocity or position,
3. The noise is non-stationary (different variance at different times).

In order to build a better intuitive understanding of what this means, let's see what this actually looks like. What I'm trying to do is use, in real time, accelerometer data from someone driving a car. Cars are interesting when dealing with this sort of data, as the way that it's forced to move due to its construction simultaneously constrains the problem (making it simpler), but adds in some unique complications. For example, let's take a look at accelerometer noise when the car is at rest compared to when it's driving at 100 km/h:

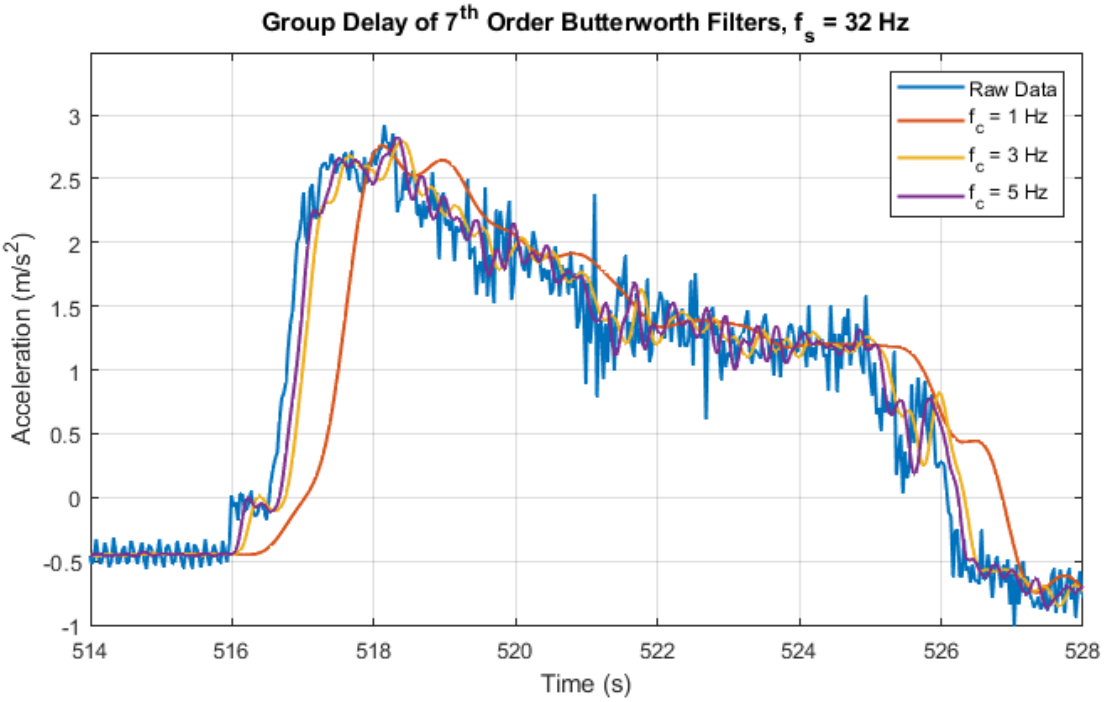


This is actually pretty interesting. First, it's hard to tell from this plot, but the noise when the car isn't moving isn't actually technically noise in the sense that it's a random, additive process. It appears to be a sinusoid because... it is. It turns out that internal combustion engines vibrate, even when the car they're powering is not moving. It's like there are internal parts that jiggle around at a few thousand rotations per minute (~50 – 100 Hz), which then gets aliased down, showing up as a roughly sinusoidal signal in our accelerometer as it's jostled around.

The second reason it's interesting is because unlike, say, thermal noise, the noise added to the accelerometer is definitely non-stationary. I mean, on top of the noise added by the car, there's also noise from moving on top of a bumpy road (especially speed bumps and potholes), as well as noise from the air, either the wind or even turbulence as you drive by a truck on the freeway.

Now, they say that typical digital signal processing techniques fail in the presence of non-stationary noise—probably because wide-sense stationarity is a required assumption for all analytical techniques—because it can mess with your power spectrum / spectral density, making it hard to pick out what's signal and not. This can get tricky when you have relatively extreme noise signals (e.g. going over a speedbump) that are in the same ballpark as a normal signal (e.g. stopping at a stop sign) frequency-wise, i.e. around 1 Hz.

When you then try to filter out that noise near where you have signal using something like a low-pass filter, additional defects can occur, such as group delay. For a seventh order Butterworth filter with different cut-off frequencies, let's see what that looks like for a small subset of sample acceleration data:



A copy of this data should be available near where this file is stored, in a MATLAB .mat file, `EMDFilterData.mat`, and is just a quick snapshot of a car starting from rest, quickly accelerating, and then slowly accelerating over the course of ten seconds, decelerating just a little bit at the end. The reason, "at rest," has a negative value for acceleration is because this data has yet to have gravity removed from it. While the filtered data, regardless of cut-off frequency (f_c), has some delay to it (shifted slightly to the right), the data filtered with a cut-off frequency of 1 Hz is shifted significantly further.

The reason the delay is so much higher at a lower cut-off frequency is because in the z-domain (assuming a bilinear transform was used), the Butterworth filter has its poles as complex conjugate pairs symmetrically placed around the real line near positive one. As the cut-off frequency approaches zero, all of these poles don't just get closer to the unit circle at positive one, but to each other. This means that the phase response due to each individual pole adds up constructively, meaning there's a huge, sharp phase shift, and thus a huge group delay. This is a fundamental property of these digital filters, and if you're only interested in very low frequencies—whether because that's where your signal is, or you want to remove DC components, using e.g. a similar Butterworth filter—you're going to have a bad time. You can add other equalization filters or whatever to compensate, but those will suffer from similar defects at low cut-off frequencies.

So now what?

We've got noise that can be signal (or vice-versa), and no matter what we do, our filtered data will be shifted in time a little bit making it impossible to use in real-time applications. Is it possible to deal with non-stationary noise and also eliminate group delay?

I'm going to say yes, and to do this we can use an interesting technique that I stumbled across for analyzing nonlinear and non-stationary data a little while ago, the Hilbert-Huang transform[1]. Similar to other approximation techniques like the Taylor series, or integral transforms like the various Fourier or Wavelet methods, the Hilbert-Huang transform (HHT) first seeks to decompose a signal into its component parts. However, instead of breaking things out in terms of polynomials, sinusoids, or other orthonormal basis sets (looking at you, overlap integral (the fiber-optics one)), the

[1] Huang, Norden, et al. "The empirical mode decomposition and the Hilbert spectrum for nonlinear and non-stationary time series analysis." Proc. R. Soc. Lond. A (1998), 454, pp. 903 – 995.

first step in the HHT is to perform Empirical Mode Decomposition (EMD) on your signal, separating it out into different Intrinsic Mode Functions (IMFs), and then whatever's left over in that process is called the residual.

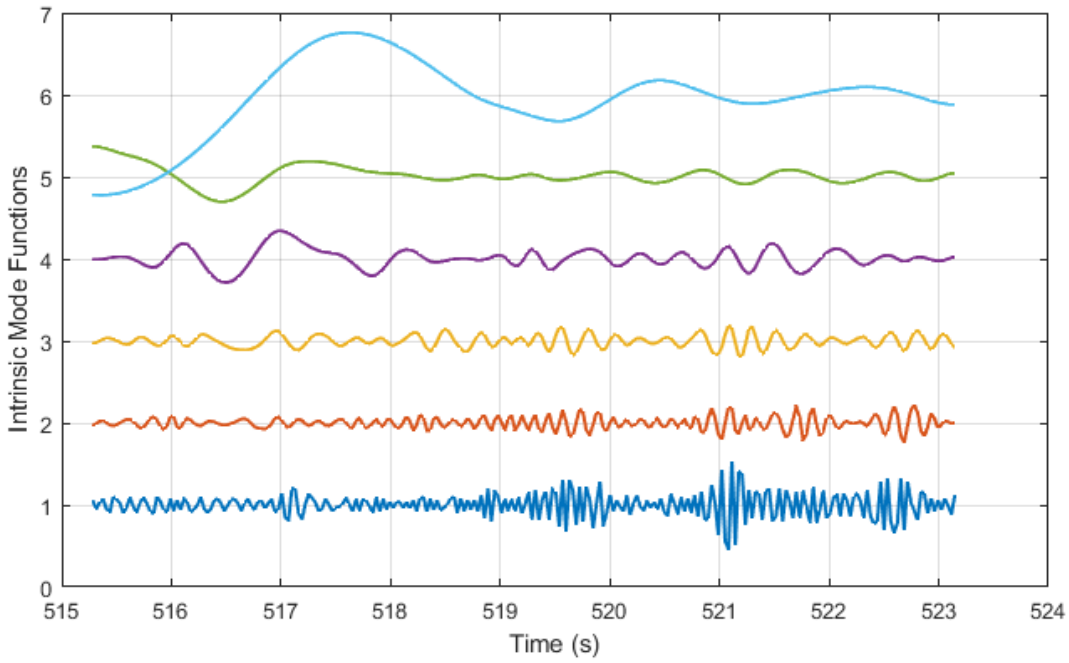
Without going through the formal definition, an IMF is like a sinusoid in that it should oscillate around zero, and if sent off to infinity will have zero mean. One way of saying this is that the number of extrema in an IMF (maxima and minima) will differ from its number of zero crossings by no more than one. To decompose a signal into its IMFs, EMD is used, which is a recursive process where we start by finding high-frequency IMFs, removing them from our signal to then find ever lower frequency IMFs until we're left with a monotonic signal (or one not meeting the definition of an IMF), which is called the residual. Overall, the best way to implement the EMD process is to follow the instructions from Wikipedia[2]:

> The procedure of extracting an IMF is called sifting. The sifting process is as follows:
>
> - Identify all the local extrema in the test data.
> - Connect all the local maxima by a cubic spline line as the upper envelope.
> - Repeat the procedure for the local minima to produce the lower envelope.
>
> The upper and lower envelopes should cover all the data between them. Their mean is m_1. The difference between the data and m_1 is the first component h_1:
>
> $$X(t) - m_1 = h_1.$$
>
> Where *X(t)* is your test data. Ideally, h_1 should satisfy the definition of an IMF, since the construction of h_1 described above should have made it symmetric and having all maxima positive and all minima negative. After the first round of sifting, a crest may become a local maximum. New extrema generated in this way actually reveal the proper modes lost in the initial examination. In the subsequent sifting process, h_1 can only be treated as a proto-IMF. In the next step, h_1 is treated as data:
>
> $$h_1 - m_{11} = h_{11}.$$
>
> After repeated sifting up to *k* times, h_1 becomes an IMF, that is
>
> $$h_{1(k-1)} - m_{1k} = h_{1k}.$$
>
> Then, h_{1k} is designated as the first IMF component of the data:
>
> $$c_1 = h_{1k}.$$
>
> This procedure is then repeated on c_1 to find the next IMF.

Sample code with my take on the sifting process of finding IMFs (including some simplifications for performance) is found in the `getimfs.m` MATLAB function; also included in the appendices. I should note that for my version of sifting to find IMFs, I set the maximum value of *k* for each IMF to be 100; however, I also stop before *k* = 100 if the standard deviation between the current proto-IMF, e.g. h_{1k}, and the previous proto-IMF, $h_{1(k-1)}$, is less than 0.3 as set out in the original paper[1] and it's at that point where h_{1k} is considered an IMF. To make sure that this process always converges, I also mirror the data before analysis, while only calculating the standard deviation for the non-mirrored section of data. This is discussed further below, and should make sense by the end!

If you take our sample data above and follow this process through to find all the possible IMFs, you get something like this:

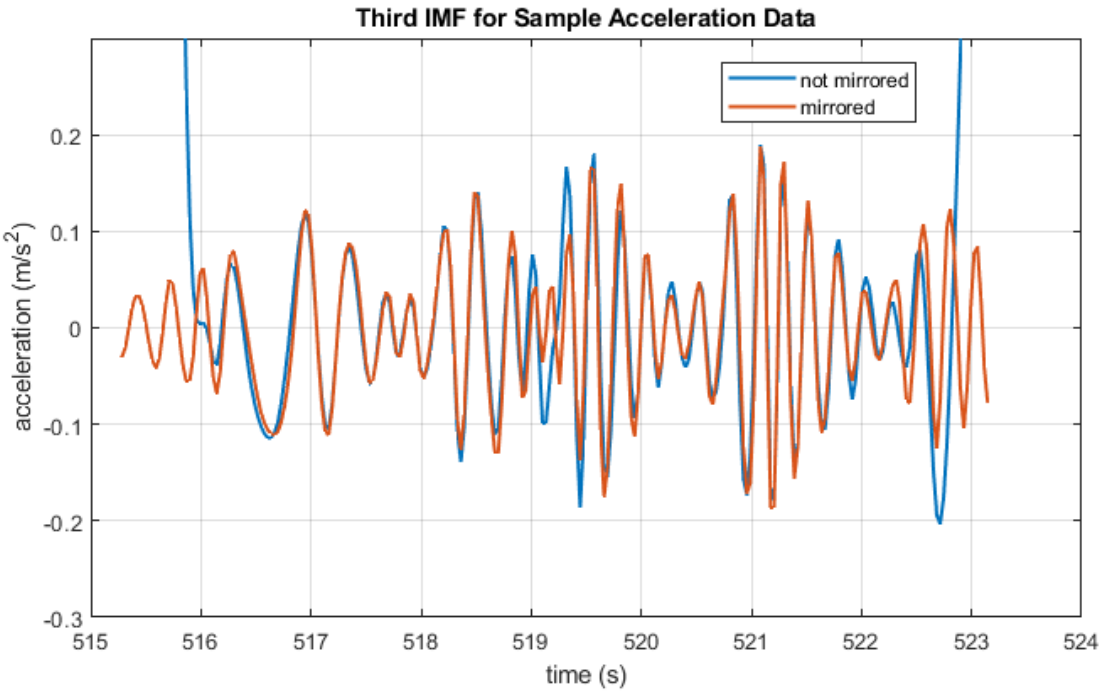
[2] https://en.wikipedia.org/wiki/Hilbert%E2%80%93Huang_transform#Empirical_mode_decomposition_(EMD)_2



Here we have the six IMFs that could be found within this sample data. Not pictured here is the residual, which contains very slowly moving signal, as well as all of the DC components of the signal. The residual is necessary to perfectly reconstruct the sample data by adding up the IMFs. Without it, the reconstructed signal would just be a weird noisy data stream centered around zero. Note that the first IMF found, number one in the above figure, contains the highest frequency parts of the signal. As we find more IMFs, they become lower in frequency, until (typically) a monotonic signal is found, i.e. the residual.

Before we continue, let's quickly discuss, "end effects," and mirroring data. Because of the way sifting works by removing higher frequency signals first, the higher order IMFs are constructed by interpolating between fewer and fewer extrema. However, because we use cubic splines to interpolate, if there are no extrema near the ends of your data at this point in the sifting process, you end up *extrapolating* using cubic splines which... tend to increase rather quickly. This is one form of, "end effect," resulting in an extreme increase of your signal as you get near either end. In order to mitigate this, one of the modifications to EMD that is discussed[3] is to mirror your data. To see what that looks like, here is an example using the third IMF that we found above:

[3] Rilling, Gabriel et al. "On Empirical Mode Decomposition and its Algorithms."



As you can see, if you mirror your data properly before you perform the EMD, it remains relatively well behaved over the region of interest. If you don't, and you do nothing to compensate for end effects, you get cubic-like increases on either end of your IMF. This can make it harder for you to find IMFs, as the standard deviation you calculate to decide when to end the sifting process for your current IMF can get stuck in an infinite loop where it never converges. Also, it's possible with very low frequency signals for these end effects to take up two thirds of your total IMF; one third on either end. Because of that, I use the following mirroring process on the original signal before I start EMD:

1. Take a copy of your original data and flip it,
2. Place this flipped copy on either end of your original data,
3. You will now be performing EMD on a signal that's three times as long,
4. Calculate the standard deviation only on the middle part, the part where your original data was.

After doing this, I find that the sifting process to extract a single IMF always converges (I have no proof of this, it's just always converged for me since I started mirroring), even for ridiculously complex and long signals (hundreds of thousands of points). Some people have recommended automatically halting the sifting process per IMF after 10 iterations because it's, "good enough," and you can't guarantee convergence, but with extreme mirroring like this, I've seen things always converge under 100 iterations—which is a lot, but nowhere near never.

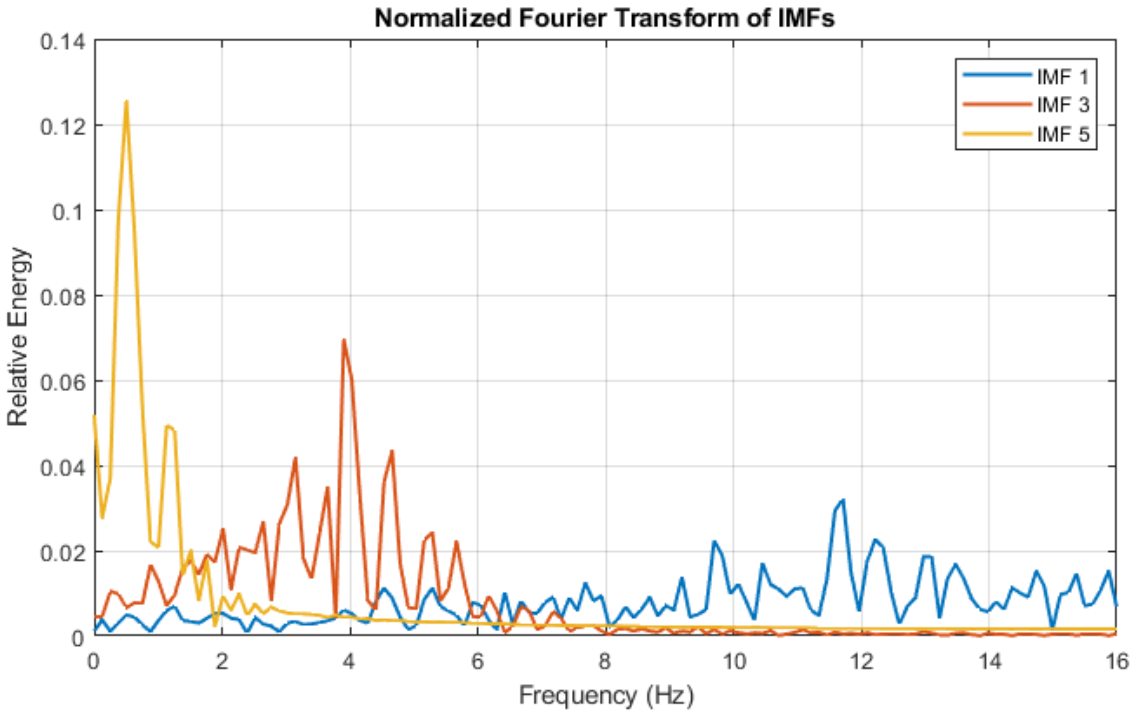
With that out of the way, how do we use this as a filter?

Well, we've already decomposed our original signal into its constituent parts, the IMFs, so all we have to do is figure out which IMFs have unwanted noise in them, and then reconstruct our signal without those IMFs. Should be easy enough. Based on the requirements for my specific problem, I know that I want to reject all frequencies higher than around 1 or 2 Hz, so all we have to do is analyze IMFs to see which ones have significant frequency components above that. The nice thing about IMFs though, is that as they're created, each IMF contains lower and lower frequency signals, so we don't have to check all of our IMFs, we can start with the lowest frequency ones, and stop when we get to the first one that contains frequencies higher than we'd like.

But how can we determine the frequency content of each IMF? Well, there are two ways to do that. Fourier transforms are obviously one way, but with the HHT, instantaneous frequency is used to better characterize the frequency content of signals as they vary in time (i.e. contrasted with a spectrogram or wavelet transform).

In the appendix (and somewhere near where this file is kept), I've included code to generate the instantaneous frequency of a signal, `getanalytic.m`, by first finding the analytic representation of that signal. The easiest way to find the analytic representation of a signal is to simply take its Fourier transform, delete all of the negative frequency components, double the energy of the positive frequency components, and then take the inverse transform. Your signal is now analytic, and complex-valued, meaning you can put it in terms of time-varying amplitude and phase. The instantaneous frequency is just the derivative of the phase with respect to time.

To get some insight into how to approach this, let's first take a look at the Fourier transform of a few of the IMFs:



This is simply the normalized, one-sided transform for three of the IMFs that we found above. Keep in mind that due to the way we collected this data via a smartphone, the original signal is not evenly spaced in time, so this transform might not be perfectly accurate. I've plotted the frequency for only these three IMFs simply because it's easier to fit their spectra all on one plot and still see what's going on. There's nothing special about this, it has all of the same features as every other discrete time Fourier Transform, such as the maximum frequency that we can represent is given by half the sample rate (32 samples / s is 16 Hz), and the resolution between frequencies is determined by the total length of the signal; so an approximately eight second signal like the one we're analyzing has a resolution about an eighth of a Hz.

It's also pretty obvious that the first IMF is mostly noise. While it definitely has more energy in the higher frequencies, part of the energy is used for frequencies that are flat across the entire spectrum, a sign of additive white Gaussian noise. To justify removing this IMF, we can use statistics to classify it—since it's been normalized—by calculating its mean, 9.7994 Hz, and its variance, 16.8439 Hz. Since it's got such a high frequency mean, and such a wide variance, we can then remove this IMF from our signal, making the claim that it is indeed mostly noise.

And this is how our filter is going to work: we look at each IMF, and make a judgement call it see if we can re-assemble our signal using only the IMFs that don't contain noise. For our specific application, we know that the dynamics of the problem we're investigating only allow the signal to really exist below 2 Hz, so it's pretty safe to reject this first IMF as noise.

Continuing on with IMF 3, this spectrum appears as though it might contain some useful signal as it looks like a frequency shifted sinc function (meaning a modulated square wave in the time domain), but since the frequency is

still relatively high, with a mean of 3.8257 Hz and variance of 5.1142 Hz, we'll still reject it as noise. That's also the irony with this current low-frequency application: true IMFs (zero mean, oscillatory signals) are in general most likely to be noise, when looked at during short time frames. Unfortunately, since we're trying to build a near real-time filter, we're kind of restricted to looking at short time frames!

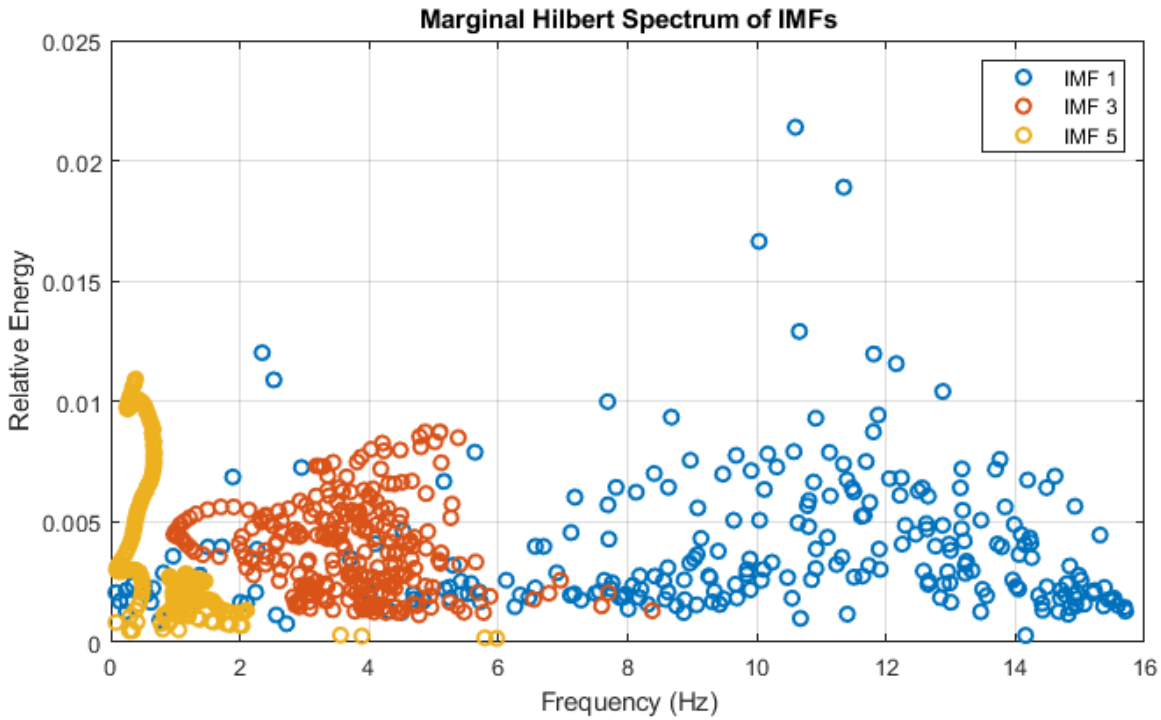
Now, looking at IMF 5, we can see that something weird is going on. This is because it starts and stops above zero (so is not zero-mean, with a small DC component), and isn't very symmetrical, so it's technically not an IMF. Again, this is a consequence of looking at a low frequency signal over a short time-frame, and means that the spectrum is subject to extreme aliasing. We can tell this because while most of the signal is less than 2 Hz (where we'd expect our true signal to be), there is an extremely long tail going off to higher frequencies, with a magnitude greater than that of IMF 3 for frequencies above 8 Hz. This significantly skews our statistics, giving us a mean frequency of 2.6931 Hz, and a variance of 14.4597 Hz! If we already hadn't spent so much talking about how all of our signal is less than 2 Hz, we might naively reject this IMF as being noise when—based on the problem we're trying to solve—we're pretty sure it isn't. Also, if you stare at IMF 5 with your bare eyes, it just looks as though the highest meaningful frequency signal present is barely above 1 Hz.

So... now what?

Well, thankfully because we're using Empirical Mode Decomposition to find Intrinsic Mode Functions our thought process is biased to immediately consider finding the instantaneous frequency in order to calculate the Hilbert spectrum, i.e. perform a full Hilbert-Huang Transform. The basic process for this can be thought of as:

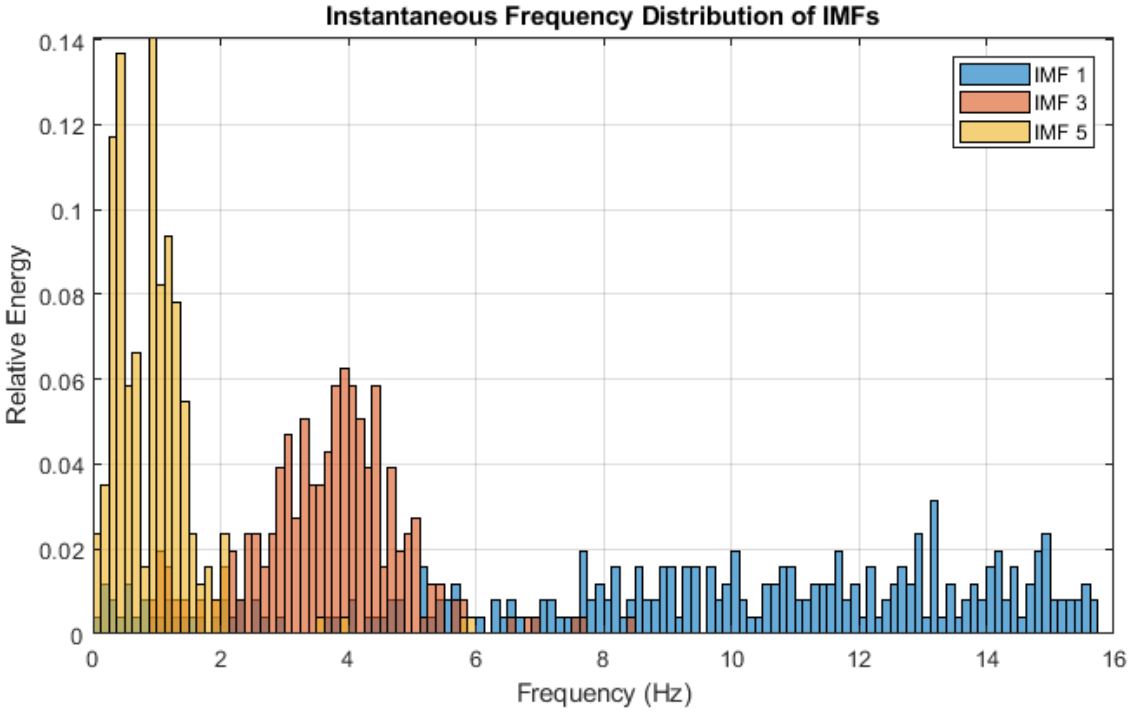
1. Perform EMD to find your IMFs,
2. Create analytic signals for each IMF,
3. Calculate the instantaneous frequency for each IMF,
4. Combine steps 2 and 3 to get your HHT.

Basically, the HHT takes the analytic representation of each IMF, and at each point in time, it says that the energy of the instantaneous frequency at that time is given by the magnitude of your analytic representation at that same time. This is now a 3D stream of data (time-frequency-energy), but we can compress it by squeezing out the time dimension to look only at frequency and energy to get the Marginal Hilbert Spectrum. Normally I'd explain this kind of thing using math, but the notation is very confusing—Wikipedia is no help—even though the concept is really quite simple. Example code to create and plot a marginal spectrum is given in the `EMD_frequency_plots.m` file in the appendix. It also creates the graph up above, and below. And this is what the marginal spectrum of our three IMFs looks like:



If you're used to staring at Fourier spectra over the course of a decade or two, this appears to be quite wacky. While we're still constrained to the same maximum frequency given by our sample rate (this is a fundamental limit after all), it appears that we're no longer constrained by frequency resolution. That is, the distance between frequencies seems to get arbitrarily small which is why the circles all bunch up. However, since the instantaneous frequency is given by the gradient of the phase of the analytic representation of the IMF this makes sense, especially for low frequency signals where the phase changes slowly with time. The resolution limit of the Fourier transform would cause all of these points to show up as energy in one signal. Although, the nice thing about this approach is that the extreme aliasing seen in IMF 5 is no longer there, so the mean and variance of the instantaneous frequency is much more reasonable.

We can also look at this in one more way, by simply plotting a histogram of the frequencies present in the instantaneous frequency. This is similar to the Marginal Hilbert Spectrum, but instead of weighting each frequency by the power given by its analytic representation, we can simply bin the raw frequencies, counting up how many are in each bin. However, because the instantaneous frequency can also be negative, we first take the absolute value of the instantaneous frequency so it's easier to compare to the spectrum from the Fourier transform. After normalizing, we can get a simple probability distribution (or relative energy distribution) of:



Now this is interesting. It looks remarkably similar to the relative energy spectra given by the Fourier transform, except again, there is no extreme aliasing in the frequency distribution of IMF 5.

Ok, but how do we characterize these distributions in order to use them in a filter? That is, how do we choose which IMFs to reject when reconstructing our signal without noise? Previously we looked at some basic statistics for the Fourier transform of each IMF, so let's continue that trend for both the marginal Hilbert spectrum and the histogram of instantaneous frequency. First, we look at the average frequency for each IMF with each method:

Mean Frequency (Hz)	IMF 1	IMF 3	IMF 5
Marginal Hilbert	9.9308	3.6199	0.6504
Instantaneous Frequency	9.5679	3.6921	0.9313
Fourier Transform	9.7994	3.8257	2.6931

The row for the mean frequencies of the Fourier Transform for each IMF is the same as discussed above, but it's interesting to note that for the higher frequency IMFs, IMF 1 and 3, the mean frequencies of any method gives similar results. However, for the lowest frequency IMF here, IMF 5, the means vary quite widely. The most obviously wrong one—even if you just eyeball the IMF and guess the mean frequency—is the result from the Fourier transform. We discussed how the extremely high mean given by the Fourier transform is due to aliasing (in this case analysing short signals for low frequencies), but what explains the discrepancy between the mean frequency calculated using the histogram of instantaneous frequency compared with the marginal Hilbert spectrum?

It simply comes down to the weighting, which is of course the definition of the marginal spectrum. When we take the histogram of the instantaneous frequency alone, we're effectively treating each frequency the same. However, for the marginal spectrum, they're weighted by the energy in the analytic representation of the frequency. But which one is, "more real," and thus more useful for our filter? In the original Huang paper, they discuss how there doesn't seem to be any physical meaning to instantaneous frequency for a signal, arguing that it can make sense for IMFs. And I buy that. This is all an empirical process, so other than the basics of information theory and digital signal processing (Nyquist frequency, etc.), there's not a lot of theory behind it, so mostly I'm going by intuition here. But while I'd say the marginal spectrum is the most accurate way to characterize low frequency IMFs, just finding the mean of the instantaneous frequencies is good enough.

Another way we can try to characterize things is through the variance in frequency of the IMFs:

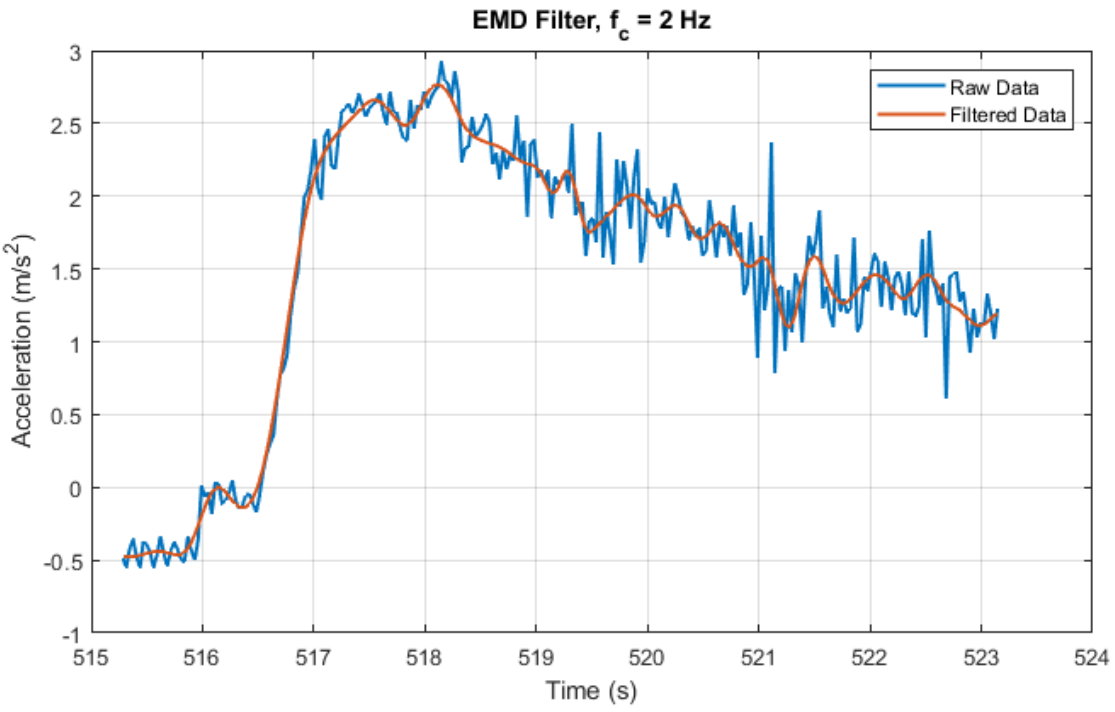
Variance (Hz^2)	IMF 1	IMF 3	IMF 5
Marginal Hilbert	14.0982	1.2746	0.1609
Instantaneous Frequency	18.3255	1.3885	0.4794
Fourier Transform	16.8439	5.1142	14.4597

Similar to its mean, the variance for IMF 5 via Fourier transform is significantly higher than we'd expect just by looking at the IMF. I mean, if you add the standard deviation to the mean, then you'd expect something like 84% of all frequencies present in IMF 5 to be about 6 Hz or less. But if you compare this to the marginal spectrum or instantaneous frequency method, they seem to give much more realistic results. However, for the sake of simplicity and relative ease of computation, I'm just going to use the mean of the instantaneous frequency to decide which IMFs to reject.

And that's it, that's our EMD filter:

- Take your original signal, decompose it into IMFs,
- Starting with the highest order (lowest frequency) IMF:
  - Find the analytic representation of the IMF,
  - Find the instantaneous frequency of the IMF,
  - Find the mean instantaneous frequency,
  - If the mean frequency is less than your cut-off frequency, that IMF can be added to your reconstructed signal,
- Repeat until you find the first IMF with a mean frequency above the cut-off,
- Check if the residual should be included as well.

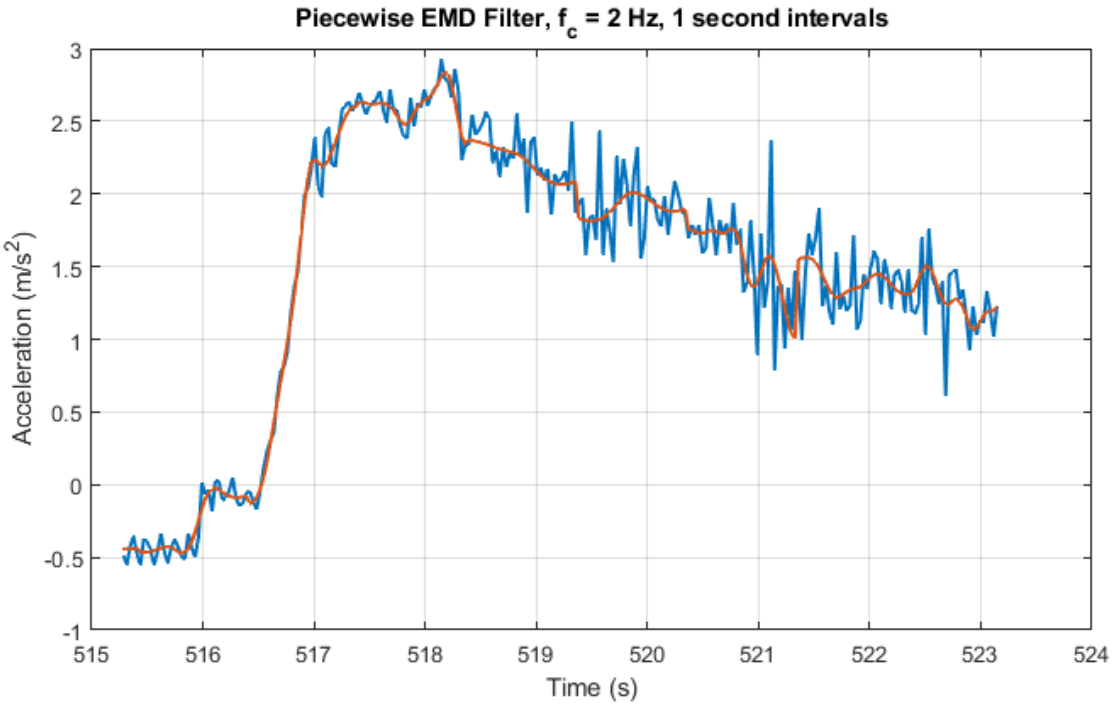
Code for a working EMD filter is given in Appendix D, and in the `emdfilter.m` file stored somewhere near where this document is kept. If we use this method across our entire signal above, the results are actually pretty good:



Using the mean instantaneous frequency to choose which IMFs should be part of our reconstructed signal, we had to pick a 2 Hz cut-off frequency, which doesn't necessarily mean that our signal has components near 2 Hz (as we saw above), but we played around to tune our filter to get decent results.

However, this is *not* anywhere near real-time. First off, we're analyzing eight seconds of signal all at once. Second, not only is EMD in general a recursive method therefore mostly single-threaded (only parts of it can be parallelized), it's not that quick to calculate. On my current machine, analyzing this eight second segment of signal can only happen about 20 times a second, which—if you need to do more math with this result—is far too slow.

So how do we make it near real-time? Well, we just analyze smaller sections at a time. If we only analyze one second intervals as we need them, we can get something closer to real-time. However, if you contrast this with, say, updating a filter every single time you get a new piece of data, then it is certainly much slower. If we stick to one second intervals, we can get a result that looks like:



Which is still pretty good! It might not be obvious, but at the edges of the one second interval, there can be discontinuities in your reconstructed signal, as noise just makes things a bit wonky no matter how you do it. Sticking to smaller intervals also means things can be filtered much faster. At one second (in this case, 32 samples), the filter can be run almost 300 times a second, so if you're normally sampling at a few tens of hertz, it's possible to weave this into a standard filtering scheme without too much extra overhead.

The one thing you do have to keep in mind is that information theory is always fundamental. In this case you always have to keep in the back of your mind that the frequency resolution is determined by the length of your signal. While we saw this statement didn't mean much when talking about instantaneous frequency, it still ultimately determines the lowest frequency IMF that you're able to find. For example, the entire eight second, 32 Hz signal we've been talking about can be decomposed into six different IMFs as well as a residual. However, when we only look at single second sections of this signal, we're only able to find three IMFs and the residual. As far as I can tell, there's no hard and fast theory on any of this, so you'll have to play around to see what works for you.

Appendix A: MATLAB Code to Find IMFs

```
%GETIMFS   Performs Empirical Mode Decomposition (EMD) on data in order to
%   compute and return the Intrinsic Mode Functions (IMFs).  Sifting is
%   based on standard deviation metric in the original by Huang et al, but
%   uses shortcuts on monotonicity and peak finding to speed things up.
%
%   imf = GETIMFS(y, x) returns the Intrinsic Mode Functions of y, a
%   function of x, sifting a maximum of 100 times to find each IMF.
%
%   imf = GETIMFS(..., maxnimf) returns up to maxnimf IMFs regardless of
%   how many there actually may be.  However, the EMD process will stop
%   whenever the residual becomes approximately monotonic.
%
%   imf = GETIMFS(..., idx) uses an index vector given by idx to find the
%   standard deviation of your IMF in the sifting process.  By default the
%   full y vector is used, but to eliminate end effects, you might want to
%   focus only on a specific region of interest, i.e. defined by idx.
%
%   [imf, r] = getimfs(y, x) also returns the residual, r.
%
%   [imf, r, nimf] = getimfs(y, x) also returns the total number of IMFs
%   found.
%
%   See also: EMDFILTER, GETANALYTIC.
function [imf, r, nimf] = getimfs(y, x, maxnimf, idx)

    useidx = false;
    if nargin == 4
        useidx = true;
    end

    wasrow = false;
    if isrow(y)
        wasrow = true;
        y = y(:);
        x = x(:);
    end

    imf = zeros(length(y), maxnimf);
    for i = 1:maxnimf
        h = y;

        extrema = conv([-0.5, 0.5], sign(diff(h)));
        idxmax = extrema == 1;
        idxmin = extrema == -1;

        maybemonotonic = (length(find(idxmax, 3)) < 3) || (length(find(idxmin, 3)) < 3);
        if maybemonotonic
            i = i - 1; %#ok<FXSET>
            break;
        end

        upper = spline(x(idxmax), h(idxmax), x);
        lower = spline(x(idxmin), h(idxmin), x);

        meanenv = (upper + lower) / 2;
        h = h - meanenv;

        for j = 1:100
            extrema = conv([-0.5, 0.5], sign(diff(h)));
            idxmax = extrema == 1;
            idxmin = extrema == -1;

            maybemonotonic = (length(find(idxmax, 3)) < 3) || (length(find(idxmin, 3)) < 3);
            if maybemonotonic
                break;
            end
```

```
            upper = spline(x(idxmax), h(idxmax), x);
            lower = spline(x(idxmin), h(idxmin), x);

            meanenv = (upper + lower) / 2;

            if useidx
                SD = sum(meanenv(idx).^2 ./ h(idx).^2);
            else
                SD = sum(meanenv.^2 ./ h.^2);
            end

            h = h - meanenv;
            if SD < 0.3
                break;
            end
        end

        imf(:,i) = h;
        y = y - h;
    end
    r = y;
    nimf = i;
    imf = imf(:,1:nimf);

    if wasrow
        imf = imf.';
        r = r.';
    end
end
```

Appendix B: MATLAB Code to Create Analytical Signals

```
%GETANALYTIC   Computes the complex, analytic representation of a
%   function, as well as its instantaneous frequency.
%
%   ya = GETANALYTIC(y, x) returns the analytic representation of y, a
%   function of x.
%
%   [ya, fi] = GETANALYTIC(y, x) also returns the instantaneous frequency,
%   fi, which is just the gradient of the angle of ya.
%
%   See also: GETEMD, EMDFILTER.
function [ya, fi] = getanalytic(y, x)

wasrow = false;
if isrow(y)
    wasrow = true;
    y = y.';
end

isodd = mod(length(y), 2) > 0;
if isodd
    Y = fft([y; y(end)]);
else
    Y = fft(y);
end
n = ceil(length(Y) / 2);

ya = ifft([Y(1); 2*Y(2:n-1); Y(n); zeros(n, 1)]);

if isodd
    ya = ya(1:end-1);
end

if nargout > 1
    phi = unwrap(angle(ya)) / 2 / pi;
    fi = gradient(phi, x);
    if wasrow
        fi = fi.';
    end
end

if wasrow
    ya = ya';
end
```

Appendix C: MATLAB Code to Plot Various Spectra

```
clear;
load EMDFilterData;

%%
[c, r] = getemd(y, x);

[x1, y1, idx] = mirrordata(x, c(:,1));
[x3, y3] = mirrordata(x, c(:,3));
[x5, y5] = mirrordata(x, c(:,5));

[ya1, fi1] = getanalytic(y1, x1);
[ya3, fi3] = getanalytic(y3, x3);
[ya5, fi5] = getanalytic(y5, x5);
edges = 0:0.125:16;

ya1 = abs(ya1(idx)) / sum(abs(ya1(idx))); fi1 = abs(fi1(idx));
ya3 = abs(ya3(idx)) / sum(abs(ya3(idx))); fi3 = abs(fi3(idx));
ya5 = abs(ya5(idx)) / sum(abs(ya5(idx))); fi5 = abs(fi5(idx));

%%
p1 = abs(fft(c(:,1))); p1 = p1(1:end/2); p1 = p1 / sum(p1);
p3 = abs(fft(c(:,3))); p3 = p3(1:end/2); p3 = p3 / sum(p3);
p5 = abs(fft(c(:,5))); p5 = p5(1:end/2); p5 = p5 / sum(p5);
f0 = linspace(0, 16, length(x)/2).';

%%
figure;
plot(fi1, ya1, 'o', fi3, ya3, 'o', fi5, ya5, 'o', 'LineWidth', 1.5);
xlabel('Frequency (Hz)'); ylabel('Relative Energy');
title('Marginal Hilbert Spectrum of IMFs');
grid on; xlim([0, 16]);
legend('IMF 1', 'IMF 3', 'IMF 5');

%%
figure;
histogram(fi1, edges, 'Normalization', 'Probability'); hold on;
histogram(fi3, edges, 'Normalization', 'Probability');
histogram(fi5, edges, 'Normalization', 'Probability'); grid on;
xlabel('Frequency (Hz)'); ylabel('Relative Energy');
title('Instantaneous Frequency Distribution of IMFs');
axis tight; xlim([0, 16]);
legend('IMF 1', 'IMF 3', 'IMF 5');

%%
figure;
plot(f0, p1, f0, p3, f0, p5, 'LineWidth', 1.5); grid on;
xlabel('Frequency (Hz)'); ylabel('Relative Energy');
title('Normalized Fourier Transform of IMFs');
legend('IMF 1', 'IMF 3', 'IMF 5');
```

Appendix D: MATLAB Code for an EMD Filter

```
%EMDFILTER   Using Empirical Mode Decomposition (EMD), filter a signal by
%   eliminating Intrinsic Mode Functions (IMFs) whose mean instantaneous
%   frequency is outside the area of interest.  The input signal is
%   mirrored on either side in order to eliminate the end effects inherent
%   in the EMD process.
%
%   f = EMDFILTER(y, x) returns the filtered signal, f, based on the input
%   signal, y, a function of x.  With no upper or lower cutoff frequencies
%   given, the defaults are a low-pass filter removing 50% of lower
%   frequency content.
%
%   f = EMDFILTER(..., fup, flo) returns the filtered signal using an upper
%   cutoff frequency of fup, and a lower cutoff frequency of flo.  All
%   units for frequency are in Hertz.
%
%   f = EMDFILTER(..., maxnimf) uses a maximum of maxnimf IMFs to filter.
%
%   [f, imf, res] = EMDFILTER(...) also returns the IMFs that were computed
%   by the EMD, imf, as well as the residual signal, res.
%
%   See also: GETIMFS, GETANALYTIC.
function [f, imf, res] = emdfilter(y, x, fup, flo, maxnimf)

narginchk(2, 5);
if nargin < 5
    maxnimf = 10;
end

% these defaults set up a low-pass filter for bottom 50% of frequencies:
if nargin < 4
    flo = 0;
end

if nargin < 3
    maxf = 1 / min(diff(x)) / 2;
    fup = maxf / 2;
end

if isrow(x)
    x = x.';
end

wasrow = false;
if isrow(y)
    wasrow = true;
    y = y.';
end

[x0, y0, idxROI] = mirrordata(x, y);

[imf, res, nimf] = getimfs(y0, x0, maxnimf, idxROI);

f = zeros(length(x), 1);
for i = nimf:-1:1
    [~, fi] = getanalytic(imf(:, i), x0);

    maxfreq = mean(abs(fi(idxROI)));
    if (maxfreq > fup)
        break;
    elseif (maxfreq < flo)
        continue;
    end
    f = f + imf(idxROI, i);
end

% do a quick check to see if the residual has useful signal:
```

```
[~, fi] = getanalytic(res, x0);

maxfreq = mean(abs(fi(idxROI)));
if (maxfreq < fup) && (maxfreq > flo)
    f = f + res(idxROI);
end

if wasrow
    f = f.';
end

if nargout > 1
    imf = imf(idxROI,1:nimf);
    if wasrow
        imf = imf.';
    end
end

if nargout > 2
    res = res(idxROI);
    if wasrow
        res = res.';
    end
end

       function [x0, y0, idxROI] = mirrordata(x, y)

           idxROI = (1:length(x)) + length(x) - 1;

           x0 = [-x(end:-1:2) + 2*x(1); x; ...
                 -x(end-1:-1:1) + 2*x(end)];

           y0 = [y(end:-1:2); y; ...
                 y(end-1:-1:1)];
       end
```