Signal Processing with SciPy: Linear Filters

Warren Weckesser*

Abstract—The SciPy library is one of the core packages of the PyData stack. It includes modules for statistics, optimization, interpolation, integration, linear algebra, Fourier transforms, signal and image processing, ODE solvers, special functions, sparse matrices, and more. In this chapter, we demonstrate many of the tools provided by the `signal` subpackage of the SciPy library for the design and analysis of linear filters for discrete-time signals, including filter representation, frequency response computation and optimal FIR filter design.

Index Terms—algorithms, signal processing, IIR filter, FIR filter

CONTENTS

Introduction

The SciPy library, created in 2001, is one of the core packages of the PyData stack. It includes modules for statistics, optimization, interpolation, integration, linear algebra, Fourier transforms, signal and image processing, ODE solvers, special functions, sparse matrices, and more.

The `signal` subpackage within the SciPy library includes tools for several areas of computation, including signal processing, interpolation, linear systems analysis and even some elementary image processing. In this Cookbook chapter, we focus on a specific subset of the capabilities of of this subpackage: the design and analysis of linear filters for discrete-time signals. These filters are commonly used in digital signal processing (DSP) for audio signals and other time series data with a fixed sample rate.

The chapter is split into two main parts, covering the two broad categories of linear filters: *infinite impulse response* (IIR) filters and *finite impulse response* (FIR) filters.

Note. We will display some code samples as transcripts from the standard Python interactive shell. In each interactive Python session, we will have executed the following without showing it:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> np.set_printoptions(precision=3, linewidth=50)
```

Also, when we show the creation of plots in the transcripts, we won't show all the other plot commands (setting labels, grid lines, etc.) that were actually used to create the corresponding figure in the paper. Complete scripts for all the figures in this paper are available in the `papers/scipy` directory at `https://github.com/pydata/pydata-cookbook/`.

IIR filters in `scipy.signal`

An IIR filter can be written as a linear recurrence relation, in which the output y_n is a linear combination of x_n, the M previous values of x and the N previous values of y:

$$a_0 y_n = \sum_{i=0}^{M} b_i x_{n-i} - \sum_{i=1}^{N} a_i y_{n-N} \tag{1}$$

(See, for example, Oppenheim and Schafer [OS], Chapter 6. Note, however, that the sign convention for `a[1], ..., a[N]` in Eqn. (1) and used in SciPy is the opposite of that used in Oppenheim and Schafer.)

By taking the z-transform of Eqn. (1), we can express the filter as

$$Y(z) = H(z)X(z) \tag{2}$$

where

$$H(z) = \frac{b_0 + b_1 z^{-1} + \cdots + b_M z^{-M}}{a_0 + a_1 z^{-1} + \cdots + a_N z^{-N}} \tag{3}$$

is the *transfer function* associated with the filter. The functions in SciPy that create filters generally set $a_0 = 1$.

Eqn. (1) is also known as an ARMA(N, M) process, where "ARMA" stands for *Auto-Regressive Moving Average*. b holds the moving average coefficients, and a holds the auto-regressive coefficients.

* *Corresponding author: warren.weckesser@gmail.com*

When $a_1 = a_2 = \cdots = a_N = 0$, the filter is a finite impulse response filter. We will discuss those later.

IIR filter representation

In this section, we discuss three representations of a linear filter:

- transfer function
- zeros, poles, gain (ZPK)
- second order sections (SOS)

SciPy also provides a state space representation, but we won't discuss that format here.

Transfer function. The transfer function representation of a filter in SciPy is the most direct representation of the data in Eqn. (1) or (3). It is two one-dimensional arrays, conventionally called `b` and `a`, that hold the coefficients of the polynomials in the numerator and denominator, respectively, of the transfer function $H(z)$.

For example, we can use the function `scipy.signal.butter` to create a Butterworth lowpass filter of order 6 with a normalized cutoff frequency of 1/8 the Nyquist frequency. The default representation created by `butter` is the transfer function, so we can use `butter(6, 0.125)`:

```
>>> from scipy.signal import butter
>>> b, a = butter(6, 0.125)
>>> b
array([  2.883e-05,   1.730e-04,   4.324e-04,
         5.765e-04,   4.324e-04,   1.730e-04,
         2.883e-05])
>>> a
array([ 1.   , -4.485,  8.529, -8.779,  5.148,
       -1.628,  0.217])
```

The representation of a filter as a transfer function with coefficients `(b, a)` is convenient and of theoretical importance, but with finite precision floating point, applying an IIR filter of even moderately large order using this format is susceptible to instability from numerical errors. Problems can arise when designing a filter of high order, or a filter with very narrow pass or stop bands.

ZPK. The ZPK representation consists of a tuple containing three items, `(z, p, k)`. The first two items, `z` and `p`, are one-dimensional arrays containing the zeros and poles, respectively, of the transfer function. The third item, `k`, is a scalar that holds the overall gain of the filter.

We can tell `butter` to create a filter using the ZPK representation by using the argument `output="zpk"`:

```
>>> z, p, k = butter(6, 0.125, output='zpk')
>>> z
array([-1., -1., -1., -1., -1., -1.])
>>> p
array([ 0.841+0.336j,  0.727+0.213j,
        0.675+0.072j,  0.675-0.072j,
        0.727-0.213j,  0.841-0.336j])
>>> k
2.8825891944002783e-05
```

A limitation of the ZPK representation of a filter is that SciPy does not provide functions that can directly apply the filter to a signal. The ZPK representation must be converted to either the SOS format or the transfer function format to actually filter a signal. If we are designing a filter using `butter` or one of the other filter design functions, we might as well create the filter in the transfer function or SOS format when the filter is created.

SOS. In the *second order sections (SOS)* representation, the filter is represented using one or more cascaded second order filters (also known as "biquads"). The SOS representation is implemented as an array with shape (n, 6), where each row holds the coefficients of a second order transfer function. The first three items in a row are the coefficients of the numerator of the biquad's transfer function, and the second three items are the coefficients of the denominator.

The SOS format for an IIR filter is more numerically stable than the transfer function format, so it should be preferred when using filters with orders beyond, say, 7 or 8, or when the bandwidth of the passband of a filter is sufficiently small. (Unfortunately, we don't have a precise specification for what "sufficiently small" is.)

A disadvantage of the SOS format is that the function `sosfilt` (at least at the time of this writing) applies an SOS filter by making multiple passes over the data, once for each second order section. Some tests with an order 8 filter show that `sosfilt(sos, x)` can require more than twice the time of `lfilter(b, a, x)`.

Here we create a Butterworth filter using the SOS representation:

```
>>> sos = butter(6, 0.125, output="sos")
>>> sos
array([[  2.883e-05,   5.765e-05,   2.883e-05,
          1.000e+00,  -1.349e+00,   4.602e-01],
       [  1.000e+00,   2.000e+00,   1.000e+00,
          1.000e+00,  -1.454e+00,   5.741e-01],
       [  1.000e+00,   2.000e+00,   1.000e+00,
          1.000e+00,  -1.681e+00,   8.198e-01]])
```

The array `sos` has shape (3, 6). Each row represents a biquad; for example, the transfer function of the biquad stored in the last row is

$$H(z) = \frac{1 + 2z^{-1} + z^{-2}}{1 - 1.681z^{-1} + 0.8198z^{-2}}$$

Converting between representations. The `signal` module provides a collection of functions for converting one representation to another:

```
sos2tf, sos2zpk, ss2tf, ss2zpk,
tf2sos, tf2zz, tf2zpk, zpk2sos, zpk2ss, zpk2tf
```

For example, `zpk2sos` converts from the ZPK representation to the SOS representation. In the following, `z`, `p` and `k` have the values defined earlier:

```
>>> from scipy.signal import zpk2sos
>>> zpk2sos(z, p, k)
array([[  2.883e-05,   5.765e-05,   2.883e-05,
          1.000e+00,  -1.349e+00,   4.602e-01],
       [  1.000e+00,   2.000e+00,   1.000e+00,
          1.000e+00,  -1.454e+00,   5.741e-01],
       [  1.000e+00,   2.000e+00,   1.000e+00,
          1.000e+00,  -1.681e+00,   8.198e-01]])
```

Limitations of the transfer function representation. Earlier we said that the transfer function representation of moderate to large order IIR filters can result in numerical problems. Here we show an example.

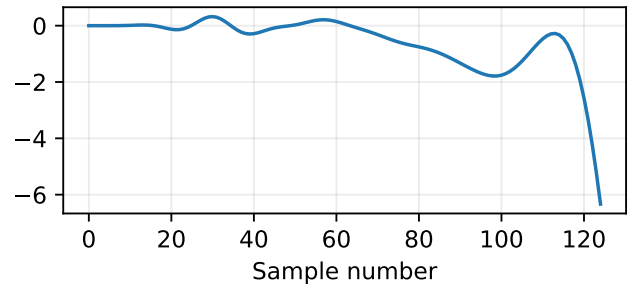

Fig. 1: *Incorrect step response of the Butterworth bandpass filter of order 10 created using the transfer function representation. Apparently the filter is unstable--something has gone wrong with this representation.*

We consider the design of a Butterworth bandpass filter with order 10 with normalized pass band cutoff frequencies of 0.04 and 0.16.:

```
>>> b, a = butter(10, [0.04, 0.16], btype="bandpass")
```

We can compute the step response of this filter by applying it to an array of ones:

```
>>> x = np.ones(125)
>>> y = lfilter(b, a, x)
>>> plt.plot(y)
```

The plot is shown in Figure 1. Clearly something is going wrong.

We can try to determine the problem by checking the poles of the filter:

```
>>> z, p, k = tf2zpk(b, a)
>>> np.abs(p)
array([ 0.955,  0.955,  1.093,  1.093,  1.101,
        1.052,  1.052,  0.879,  0.879,  0.969,
        0.969,  0.836,  0.836,  0.788,  0.788,
        0.744,  0.744,  0.725,  0.725,  0.723])
```

The filter should have all poles inside the unit circle in the complex plane, but in this case five of the poles have magnitude greater than 1. This indicates a problem, which could be in the result returned by `butter`, or in the conversion done by `tf2zpk`. The plot shown in Figure 1 makes clear that *something* is wrong with the coefficients in `b` and `a`.

Let's design the same 10th order Butterworth filter as above, but in the SOS format:

```
>>> sos = butter(10, [0.04, 0.16],
...              btype="bandpass", output="sos")
```

In this case, all the poles are within the unit circle:

```
>>> z, p, k = sos2zpk(sos)
>>> np.abs(p)
array([ 0.788,  0.788,  0.8  ,  0.8  ,  0.818,
        0.818,  0.854,  0.854,  0.877,  0.877,
        0.903,  0.903,  0.936,  0.936,  0.955,
        0.955,  0.964,  0.964,  0.988,  0.988])
```

We can check the frequency response using `scipy.signal.sosfreqz`:

```
>>> w, h = sosfreqz(sos, worN=8000)
>>> plt.plot(w/np.pi, np.abs(h))
[<matplotlib.lines.Line2D at 0x109ae9550>]
```

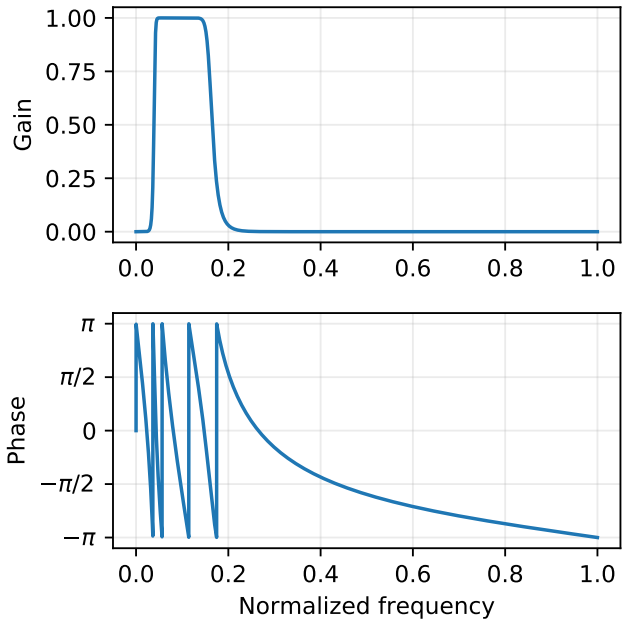

Fig. 2: *Frequency response of the Butterworth bandpass filter with order 10 and normalized cutoff frequencies 0.04 and 0.16.*

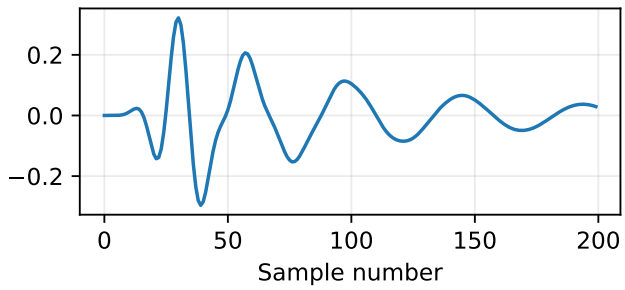

Fig. 3: *Step response of the Butterworth bandpass filter with order 10 and normalized cutoff frequencies 0.04 and 0.16.*

The plot is shown in Figure 2.

As above, we compute the step response by filtering an array of ones:

```
>>> x = np.ones(200)
>>> y = sosfilt(sos, x)
>>> plt.plot(y)
```

The plot is shown in Figure 3. With the SOS representation, the filter behaves as expected.

In the remaining examples of IIR filtering, we will use only the SOS representation.

Lowpass filter

Figure 4 shows a times series containing pressure measurements [SO]. At some point in the interval $20 < t < 22$, an event occurs in which the pressure jumps and begins oscillating around a "center". The center of the oscillation decreases and appears to level off.

We are not interested in the oscillations, but we are interested in the mean value around which the signal is

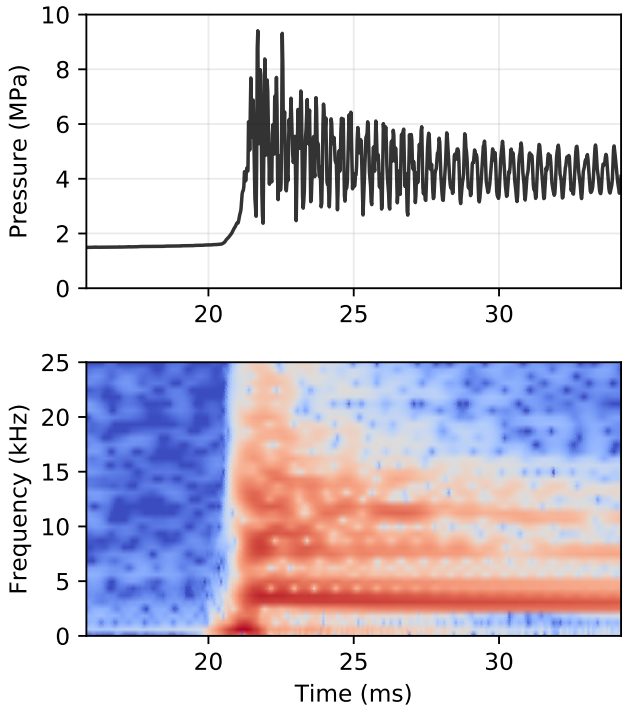

Fig. 4: Top*: Pressure, for the interval 15 < t < 35 (milliseconds).* Bottom*: Spectrogram of the pressure time series (generated using a window size of 1.6 milliseconds).*

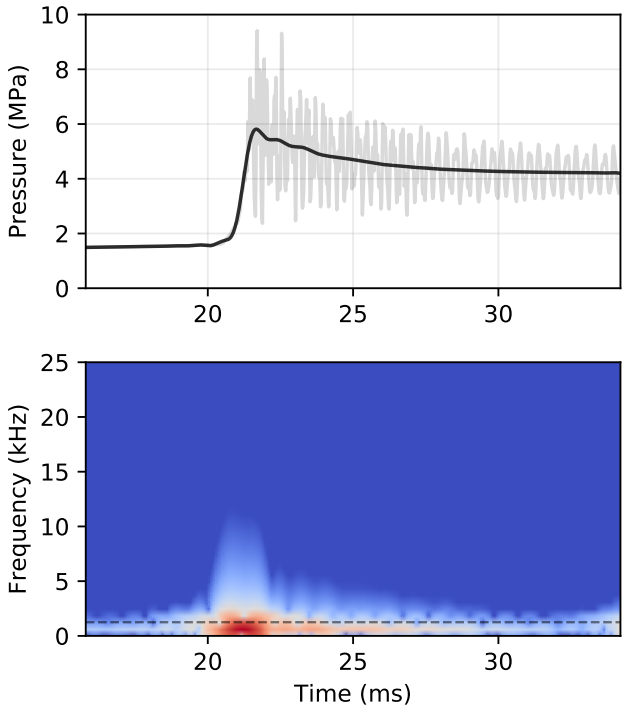

Fig. 5: Top*: Filtered pressure, for the interval 15 < t < 35 (milliseconds). The light gray curve is the unfiltered data.* Bottom*: Spectrogram of the filtered time series (generated using a window size of 1.6 milliseconds). The dashed line is at 1250 Hz.*

oscillating. To preserve the slowly varying behavior while eliminating the high frequency oscillations, we'll apply a low-pass filter. To apply the filter, we can use either `sosfilt` or `sosfiltfilt` from `scipy.signal`. The function `sosfiltfilt` is a forward-backward filter--it applies the filter twice, once forward and once backward. This effectively doubles the order of the filter, and results in zero phase shift. Because we are interesting in the "event" that occurs in 20 < t < 22, it is important to preserve the phase characteristics of the signal, so we use `sosfiltfilt`.

The following code snippet defines two convenience functions. These functions allow us to specify the sampling frequency and the lowpass cutoff frequency in whatever units are convenient. They take care of scaling the values to the units expected by `scipy.signal.butter`.

```
from scipy.signal import butter, sosfiltfilt

def butter_lowpass(cutoff, fs, order):
    normal_cutoff = cutoff / (0.5*fs)
    sos = butter(order, normal_cutoff,
                 btype='low', output='sos')
    return sos

def butter_lowpass_filtfilt(data, cutoff, fs,
                            order):
    sos = butter_lowpass(cutoff, fs, order=order,
                         output='sos')
    y = sosfiltfilt(sos, data)
    return y
```

The results of filtering the data using `sosfiltfilt` are shown in Figure 5.

Comments on creating a spectrogram. A spectrogram is a plot of the power spectrum of a signal computed repeatedly over a sliding time window. The spectrograms in Figures 4 and 5 were created using `spectrogram` from `scipy.signal` and `pcolormesh` from `matplotlib.pyplot`. The function `spectrogram` has several options that control how the spectrogram is computed. It is quite flexible, but obtaining a plot that effectively illustrates the time-varying spectrum of a signal sometimes requires experimentation with the parameters. In keeping with the "cookbook" theme of this book, we include here the details of how those plots were generated.

Here is the essential part of the code that computes the spectrograms. `pressure` is the one-dimensional array of measured data.

```
fs = 50000
nperseg = 80
noverlap = nperseg - 4
f, t, spec = spectrogram(pressure, fs=fs,
                         nperseg=nperseg,
                         noverlap=noverlap,
                         window='hann')
```

The spectrogram for the filtered signal is computed with the same arguments:

```
f, t, filteredspec = spectrogram(pressure_filtered,
                                 ...)
```

Notes:

- `fs` is the sample rate, in Hz.
- `spectrogram` computes the spectrum over a sliding

segment of the input signal. `nperseg` specifies the number of time samples to include in each segment. Here we use 80 time samples (1.6 milliseconds). This is smaller than the default of 256, but it provides sufficient resolution of the frequency axis for our plots.

- `noverlap` is the length (in samples) of the overlap of the segments over which the spectrum is computed. We use `noverlap = nperseq - 4`; in other words, the window segments slides only four time samples (0.08 milliseconds). This provides a fairly fine resolution of the time axis.
- The spectrum of each segment of the input is computed after multiplying it by a window function. We use the Hann window.

The function `spectrogram` computes the data to be plotted. Next, we show the code that plots the spectrograms shown in Figures 4 and 5. First we convert the data to decibels:

```
spec_db = 10*np.log10(spec)
filteredspec_db = 10*np.log10(filteredspec)
```

Next we find the limits that we will use in the call to `pcolormesh` to ensure that the two spectrograms use the same color scale. `vmax` is the overall max, and `vmin` is set to 80 dB less than `vmax`. This will suppress the very low amplitude noise in the plots.

```
vmax = max(spec_db.max(), filteredspec_db.max())
vmin = vmax - 80.0
```

Finally, we plot the first spectrogram using `pcolormesh()`:

```
cmap = plt.cm.coolwarm
plt.pcolormesh(1000*t, f/1000, spec_db,
               vmin=vmin, vmax=vmax,
               cmap=cmap, shading='gouraud')
```

An identical call of `pcolormesh` with `filteredspec_db` generates the spectrogram in Figure 5.

Initializing a lowpass filter

By default, the initial state of an IIR filter as implemented in `lfilter` or `sosfilt` is all zero. If the input signal does not start with values that are zero, there will be a transient during which the filter's internal state "catches up" with the input signal.

Here is an example. The script generates the plot shown in Figure 6.

```
import numpy as np
from scipy.signal import butter, sosfilt, sosfilt_zi
import matplotlib.pyplot as plt

n = 101
t = np.linspace(0, 1, n)
np.random.seed(123)
x = 0.45 + 0.1*np.random.randn(n)

sos = butter(8, 0.125, output='sos')

# Filter using the default initial conditions.
y = sosfilt(sos, x)

# Filter using the state for which the output
# is the constant x[:4].mean() as the initial
# condition.
```

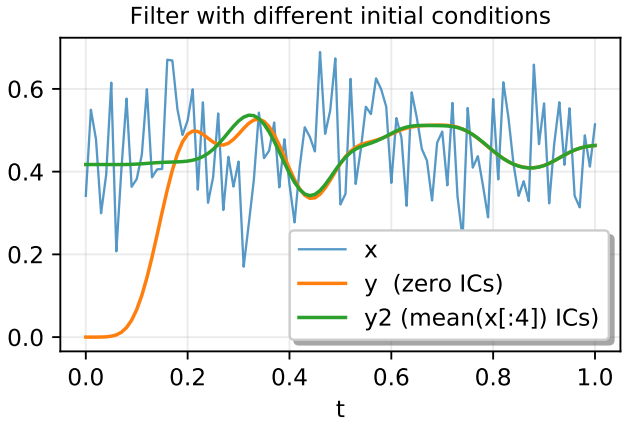

Fig. 6: *A demonstration of two different sets of initial conditions for a lowpass filter. The orange curve is the output of the filter with zero initial conditions. The green curve is the output of the filter initialized with a state associated with the mean of the first four values of the input x.*

```
zi = x[:4].mean() * sosfilt_zi(sos)
y2, zo = sosfilt(sos, x, zi=zi)

# Plot everything.
plt.plot(t, x, alpha=0.75, linewidth=1, label='x')
plt.plot(t, y, label='y  (zero ICs)')
plt.plot(t, y2, label='y2 (mean(x[:4]) ICs)')

plt.legend(framealpha=1, shadow=True)
plt.grid(alpha=0.25)
plt.xlabel('t')
plt.title('Filter with different '
          'initial conditions')
plt.show()
```

By setting `zi=x[:4].mean() * sosfilt_zi(sos)`, we are, in effect, making the filter start out as if it had been filtering the constant `x[:4].mean()` for a long time. There is still a transient associated with this assumption, but it is usually not as objectionable as the transient associated with zero initial conditions.

This initialization is usually not needed for a bandpass or highpass filter. Also, the forward-backward filters implemented in `filtfilt` and `sosfiltfilt` already have options for controlling the initial conditions of the forward and backward passes.

Bandpass filter

In this example, we will use synthetic data to demonstrate a bandpass filter. We have 0.03 seconds of data sampled at 4800 Hz. We want to apply a bandpass filter to remove frequencies below 400 Hz or above 1200 Hz.

Just like we did for the lowpass filter, we define two functions that allow us to create and apply a Butterworth bandpass filter with the frequencies given in Hz (or any other units). The functions take care of scaling the values to the normalized range expected by `scipy.signal.butter`.

```
from scipy.signal import butter, sosfilt

def butter_bandpass(lowcut, highcut, fs, order):
```

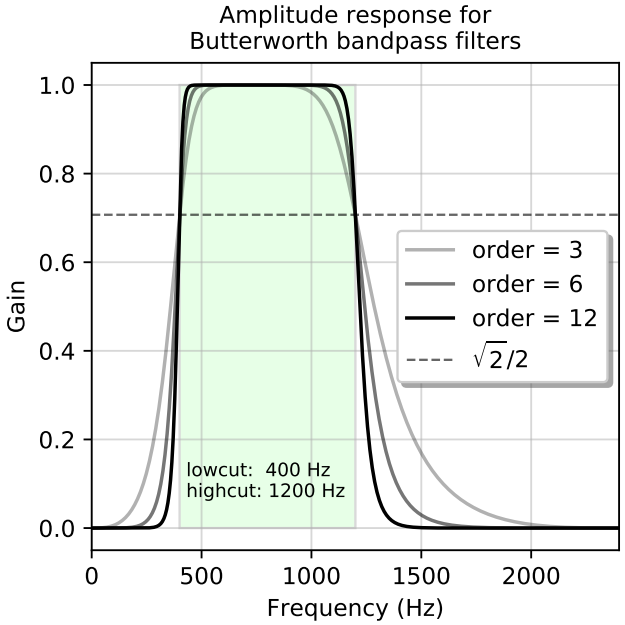

Fig. 7: *Amplitude response for a Butterworth bandpass filter with several different orders.*

```
    nyq = 0.5 * fs
    low = lowcut / nyq
    high = highcut / nyq
    sos = butter(order, [low, high], btype='band',
                 output='sos')
    return sos

def butter_bandpass_filt(data, lowcut, highcut,
                         fs, order):
    sos = butter_bandpass(lowcut, highcut, fs,
                          order)
    y = sosfilt(sos, data)
    return y
```

First, we'll take a look at the frequency response of the Butterworth bandpass filter with order 3, 6, and 12. The code that generates Figure 7 demonstrates the use of `scipy.signal.sosfreqz`:

```
for order in [3, 6, 12]:
    sos = butter_bandpass(lowcut, highcut, fs, order)
    w, h = sosfreqz(sos, worN=2000)
    plt.plot((fs*0.5/np.pi)*w, abs(h), 'k',
             alpha=(order+1)/13,
             label="order = %d" % order)
```

Figure 8 shows the input signal and the filtered signal. The order 12 bandpass Butterworth filter was used. The plot shows the input signal x; the filtered signal was generated with

```
y = butter_bandpass_filt(x, lowcut, highcut, fs,
                         order=12)
```

where `fs = 4800`, `lowcut = 400` and `highcut = 1200`.

Filtering a long signal in batches

The function `lfilter` applies a filter to an array that is stored in memory. Sometimes, however, the complete signal

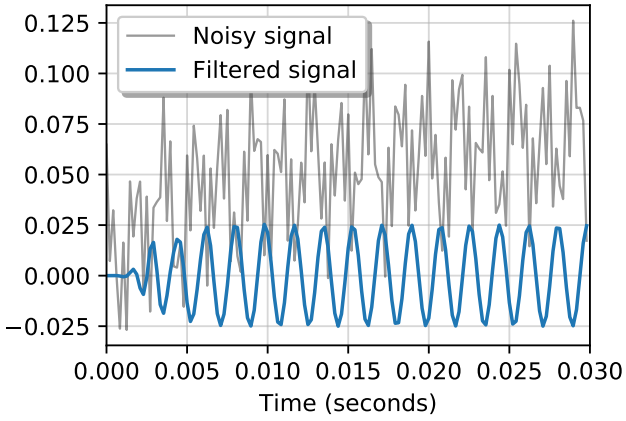

Fig. 8: *Original noisy signal and the filtered signal. The order 12 Butterworth bandpass filter shown in Figure 7 was used.*

to be filtered is not available all at once. It might not fit in memory, or it might be read from an instrument in small blocks and it is desired to output the filtered block before the next block is available. Such a signal can be filtered in batches, but the state of the filter at the end of one batch must be saved and then restored when `lfilter` is applied to the next batch. Here we show an example of how the `zi` argument of `lfilter` allows the state to be saved and restored. We will again use synthetic data generated by the same function used in the previous example, but for a longer time interval.

A pattern that can be used to filter an input signal `x` in batches is shown in the following code. The filtered signal is stored in `y`. The array `sos` contains the filter in SOS format, and is presumed to have already been created.

```
batch_size = N  # Number of samples per batch

# Array of initial conditions for the SOS filter.
z = np.zeros((sos.shape[0], 2))

# Preallocate space for the filtered signal.
y = np.empty_like(x)

start = 0
while start < len(x):
    stop = min(start + batch_size, len(x))
    y[start:stop], z = sosfilt(sos, x[start:stop],
                               zi=z)
    start = stop
```

In this code, the next batch of input is fetched by simply indexing `x[start:stop]`, and the filtered batch is saved by assigning it to `y[start:stop]`. In a more realistic batch processing system, the input might be fetched from a file, or directly from an instrument, and the output might be written to another file, or handed off to another process as part of a batch processing pipeline.

Solving linear recurrence relations

Variations of the question:

```
How do I speed up the following calculation?

y[i+1] = alpha*y[i] + c*x[i]
```

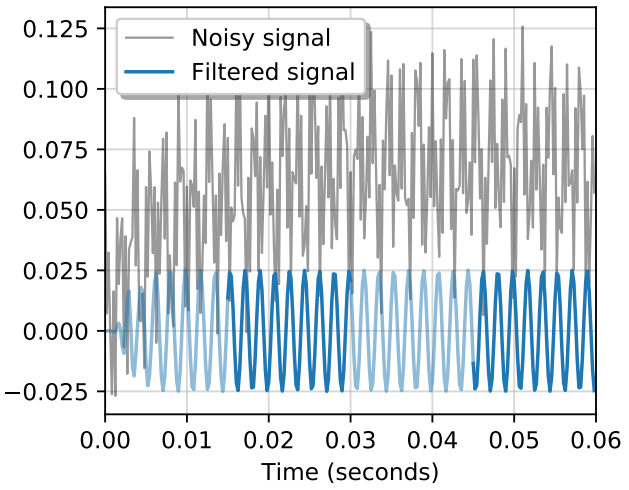

Fig. 9: *Original noisy signal and the filtered signal. The order 12 Butterworth bandpass filter shown in Figure 7 was used. The signal was filtered in batches of size 72 samples (0.015 seconds). The alternating light and dark blue colors of the filtered signal indicate batches that were processed in separate calls to* `sosfilt`.

often arise on mailing lists and online forums. Sometimes more terms such as `beta*y[i-1]` or `d*x[i-1]` are included on the right. These recurrence relations show up in, for example, GARCH models and other linear stochastic models. Such a calculation can be written in the form of Eqn. (1), so a solution can be computed using `lfilter`.

Here's an example that is similar to several questions that have appeared on the programming Q&A website `stackoverflow.com`. The one-dimensional array `h` is an input, and `alpha`, `beta` and `gamma` are constants:

```
y = np.empty(len(h))
y[0] = alpha
for i in np.arange(1, len(h)):
    y[i] = alpha + beta*y[i-1] + gamma*h[i-1]
```

To use `lfilter` to solve the problem, we have to translate the linear recurrence:

```
y[i] = alpha + beta*y[i-1] + gamma*h[i-1]
```

into the form of Eqn. (1), which will give us the coefficients `b` and `a` of the transfer function. Define:

```
x[i] = alpha + gamma*h[i]
```

so the recurrence relation is:

```
y[i] = x[i-1] + beta*y[i-1]
```

Compare this to Eqn. (1); we see that $a_0 = 1$, $a_1 = -\text{beta}$, $b_0 = 0$ and $b_1 = 1$. So we have our transfer function coefficients:

```
b = [0, 1]
a = [1, -beta]
```

We also have to ensure that the initial condition is set correctly to reproduce the desired calculation. We want the initial condition to be set as if we had values `x[-1]` and `y[-1]`, and `y[0]` is computed using the recurrence relation. Given the above recurrence relation, the formula for `y[0]` is:

```
y[0] = x[-1] + beta*y[-1]
```

We want `y[0]` to be `alpha`, so we'll set `y[-1] = 0` and `x[-1] = alpha`. To create initial conditions for `lfilter` that will set up the filter to act like it had just operated on those previous values, we use `scipy.signal.lfiltic`:

```
zi = lfiltic(b, a, y=[0], x=[alpha])
```

The `y` and `x` arguments are the "previous" values that will be used to set the initial conditions. In general, one sets `y=[y[-1], y[-2], ..]` and `x=[x[-1], x[-2], ...]`, giving as many values as needed to determine the initial condition for `lfilter`. In this example, we have just one previous value for `y` and `x`.

Putting it all together, here is the code using `lfilter` that replaces the for-loop shown above:

```
b = [0, 1]
a = [1, -beta]
zi = lfiltic(b, a, y=[0], x=[alpha])
y, zo = lfilter(b, a, alpha + gamma*h, zi=zi)
```

FIR filters in `scipy.signal`

A finite impulse response filter is basically a weighted moving average. Given an input sequence x_n and the $M+1$ filter coefficients $\{b_0, \ldots, b_M\}$, the filtered output y_n is computed as the discrete convolution of x and b:

$$y_n = \sum_{i=0}^{M} b_i x_{n-i} = (b * x)_n \tag{4}$$

where $*$ is the convolution operator. M is the *order* of the filter; a filter with order M has $M+1$ coefficients. It is common to say that the filter has $M+1$ *taps*.

Apply a FIR filter

To apply a FIR filter to a signal, we can use `scipy.signal.lfilter` with the denominator set to the scalar 1, or we can use one of the convolution functions available in NumPy or SciPy, such as `scipy.signal.convolve`. For a signal $\{x_0, x_1, \ldots, x_{S-1}\}$ of finite length S, Eq. (4) doesn't specify how to compute the result for $n < M$. The convolution functions in NumPy and SciPy have an option called `mode` for specifying how to handle this. For example, `mode='valid'` only computes output values for which all the values of x_i in Eq. 4 are defined, and `mode='same'` in effect pads the input array x with zeros so that the output is the same length as the input. See the docstring of `numpy.convolve` or `scipy.signal.convolve` for more details.

For example,

```
from scipy.signal import convolve

# Make a signal to be filtered.
np.random.seed(123)
x = np.random.randn(50)
# taps is the array of FIR filter coefficients.
taps = np.array([ 0.0625,  0.25  ,  0.375 ,
                  0.25  ,  0.0625])
# Filtered signal. y has the same length as x.
y = convolve(x, taps, mode='same')
```

There are also convolution functions in `scipy.ndimage`. The function `scipy.ndimage.convolve1d` provides an

`axis` argument, which allows all the signals stored in one axis of a multidimensional array to be filtered with one call. For example,

```
from scipy.ndimage import convolve1d

# Make an 3-d array containing 1-d signals
# to be filtered.
x = np.random.randn(3, 5, 50)
# Apply the filter along the last dimension.
y = convolve1d(x, taps, axis=-1)
```

Note that `scipy.ndimage.convolve1d` has a different set of options for its `mode` argument. Consult the docstring for details.

Specialized functions that are FIR filters

The uniform filter and the Gaussian filter implemented in `scipy.ndimage` are FIR filters. In the case of one-dimensional time series, the specific functions are `uniform_filter1d` and `gaussian_filter1d`.

The Savitzky-Golay filter [SavGol] is also a FIR filter. In the module `scipy.signal`, SciPy provides the function `savgol_coeffs` to create the coefficients of a Savitzy-Golay filter. The function `savgol_filter` applies the Savitzky-Golay filter to an input signal without returning the filter coefficients.

FIR filter frequency response

The function `scipy.signal.freqz` computes the frequency response of a linear filter represented as a transfer function. This class of filters includes FIR filters, where the representation of the numerator of the transfer function is the array of taps and the denominator is the scalar $a_0 = 1$.

As an example, we'll compute the frequency response of a uniformly weighted moving average. For a moving average of length n, the coefficients in the FIR filter are simply $1/n$. Translated to NumPy code, we have `taps = np.full(n, fill_value=1.0/n)`.

The response curves in Figure 10 were generated with this code:

```
for n in [3, 7, 21]:
    taps = np.full(n, fill_value=1.0/n)
    w, h = freqz(taps, worN=2000)
    plt.plot(w, abs(h), label="n = %d" % n)
```

The function `freqz` returns the frequencies in units of radians per sample, which is why the values on the abscissa in Figure 10 range from 0 to π. In calculations where we have a given sampling frequency f_s, we usually convert the frequencies returned by `freqz` to dimensional units by multiplying by $f_s/(2\pi)$.

FIR filter design

We'll demonstrate how SciPy can be used to design a FIR filter using the following four methods.

- *The window method.* The filter is designed by computing the impulse response of the desired ideal filter and then multiplying the coefficients by a window function.
- *Least squares design.* The weighted integral of the squared frequency response error is minimized.

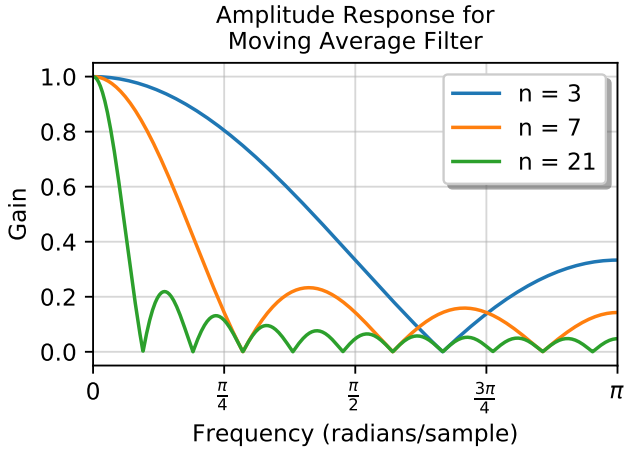

Fig. 10: *Frequency response of a simple moving average. `n` is the number of taps (i.e. the length of the sliding window).*

- *Parks-McClellan equiripple design.* A "minimax" method, in which the maximum deviation from the desired response is minimized.
- *Linear programming.* The "minimax" design problem can be formulated as a linear programming problem.

In the following sections, we discuss each design method. For this discussion, we define the following functions, where ω is the frequency in radians per sample: $A(\omega)$, the filter's (real, signed) frequency response; $D(\omega)$, the desired frequency response of the filter; and $W(\omega)$, the weight assigned to the response error at ω (i.e. how "important" is the error $A(\omega) - D(\omega)$).

FIR filter design: the window method

The window method for designing a FIR filter is to compute the filter coefficients as the impulse response of the desired ideal filter, and then multiply the coefficents by a window function to both truncate the set of coefficients (thus making a *finite* impulse response filter) and to shape the actual filter response. Most textbooks on digital signal processing include a discussion of the method; see, for example, Section 7.5 of Oppenheim and Schafer [OS].

Two functions in the module `scipy.signal` implement the window method, `firwin` and `firwin2`. Here we'll show an example of `firwin2`. We'll use `firwin` when we discuss the Kaiser window method.

We'll design a filter with 185 taps for a signal that is sampled at 2000 Hz. The filter is to be lowpass, with a *linear* transition from the pass band to the stop band over the range 150 Hz to 175 Hz. We also want a notch in the pass band between 48 Hz and 72 Hz, with sloping sides, centered at 60 Hz where the desired gain is 0.1. The dashed line in Figure 12 shows the desired frequency response.

To use `firwin2`, we specify the desired response at the endpoints of a piecewise linear profile defined over the frequency range [0, 1000] (1000 Hz is the Nyquist frequency).

```
freqs = [0, 48,  60, 72, 150, 175, 1000]
gains = [1,  1, 0.1,  1,   1,   0,    0]
```

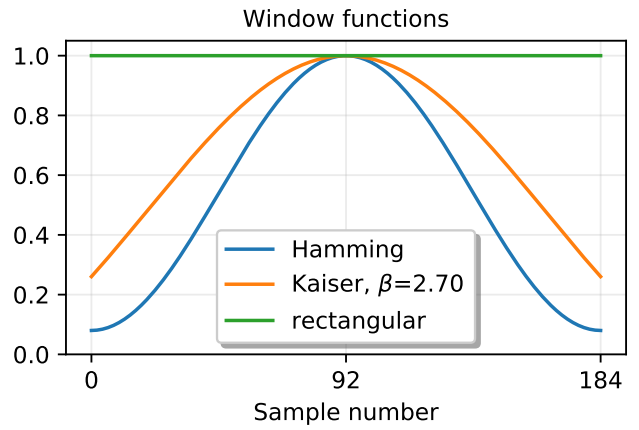

Fig. 11: *Window functions used in the* `firwin2` *filter design example.*

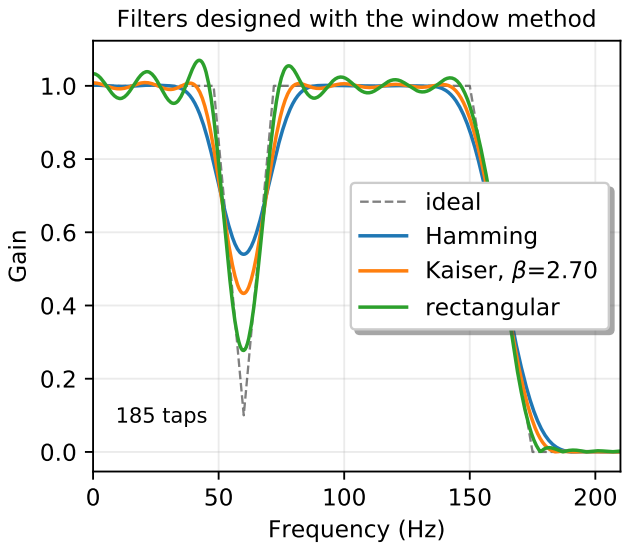

Fig. 12: *Frequency response for a filter designed using* `firwin2` *with several windows. The ideal frequency response is a lowpass filter with a ramped transition starting at 150 Hz. There is also a notch with ramped transitions centered at 60 Hz.*

To illustrate the affect of the window on the filter, we'll demonstrate the design using three different windows: the Hamming window, the Kaiser window with parameter β set to 2.70, and the rectangular or "boxcar" window (i.e. simple truncation without tapering).

The code to generate the FIR filters is

```
fs = 2000
numtaps = 185

# window=None is equivalent to using the
# rectangular window.
taps_none = firwin2(numtaps, freqs, gains,
                    nyq=0.5*fs, window=None)
# The default window is Hamming.
taps_h = firwin2(numtaps, freqs, gains,
                 nyq=0.5*fs)
beta = 2.70
taps_k = firwin2(numtaps, freqs, gains,
                 nyq=0.5*fs, window=('kaiser', beta))
```

Figure 12 shows the frequency response of the three filters.

FIR filter design: least squares

The weighted least squares method creates a filter for which the expression

$$\int_0^\pi W(\omega)\,(A(\omega) - D(\omega))^2\,d\omega \tag{5}$$

is minimized. The function `scipy.signal.firls` implements this method for piecewise linear desired response $D(\omega)$ and piecewise constant weight function $W(\omega)$. Three arguments (one optional) define the shape of the desired response: `bands`, `desired` and (optionally) `weights`.

The argument `bands` is sequence of frequency values with an even length. Consecutive pairs of values define the bands on which the desired response is defined. The frequencies covered by `bands` does not have to include the entire spectrum from 0 to the Nyquist frequency. If there are gaps, the response in the gap is ignored (i.e. the gaps are "don't care" regions).

The `desired` input array defines the amplitude of the desired frequency response at each point in `bands`.

The `weight` input, if given, must be an array with half the length of `bands`. The values in `weight` define the weight of each band in the objective function. A weight of 0 means the band does not contribute to the result at all--it is equivalent to leaving a gap in `bands`.

As an example, we'll design a filter for a signal sampled at 200 Hz. The filter is a lowpass filter, with pass band [0, 15] and stop band [30, 100], and we want the gain to vary linearly from 1 down to 0 in the transition band [15, 30]. We'll design a FIR filter with 43 taps.

We create the arrays `bands` and `desired` as described above:

```
bands =   np.array([0, 15, 15, 30, 30, 100])
desired = np.array([1,  1,  1,  0,  0,   0])
```

Then we call `firls`:

```
numtaps = 43
taps1 = firls(numtaps, bands, desired, nyq=100)
```

The frequency response of this filter is the blue curve in Figure 13.

By default, the `firls` function weights the bands uniformly (i.e. $W(\omega) \equiv 1$ in Eqn. (5)). The `weights` argument can be used to control the weight $W(\omega)$ on each band. The argument must be a sequence that is half the length of `bands`. That is, only piecewise constant weights are allowed.

Here we rerun `firls`, giving the most weight to the pass band and the least weight to the transition band:

```
wts = [100, .01, 1]
taps2 = firls(numtaps, bands, desired, nyq=100,
              weight=wts)
```

The frequency response of this filter is the orange curve in Figure 13. As expected, the frequency response now deviates

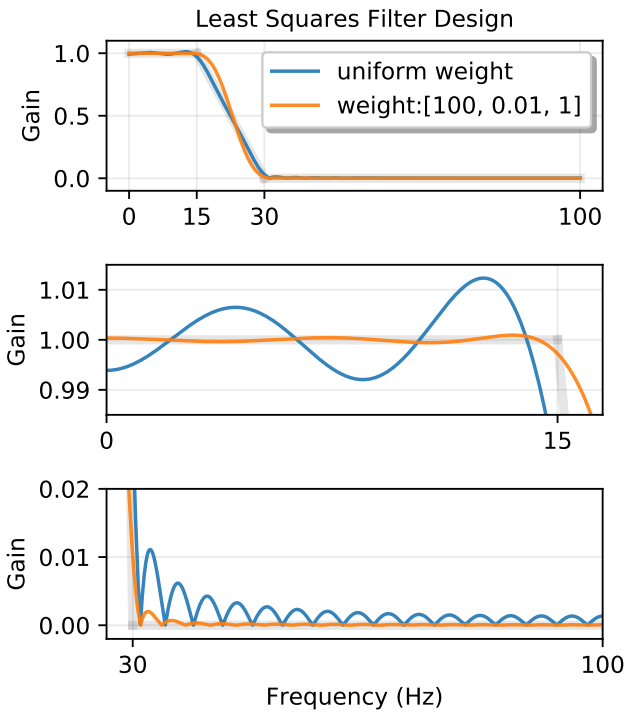

Fig. 13: *Result of a least squares FIR filter design. The desired frequency response comprises three bands. On [0, 15], the desired gain is 1 (a pass band). On [15, 30], the desired gain decreases linearly from 1 to 0. The band [30, 100] is a stop band, where the desired gain is 0. The filters have 43 taps. The middle and bottom plots are details from the top plot.*

more from the desired gain in the transition band, and the ripple in the pass band is significantly reduced. The rejection in the stop band is also improved.

Equivalence of least squares and the window method.

When uniform weights are used, and the desired result is specified for the complete interval $[0, \pi]$, the least squares method is equivalent to the window method with no window function (i.e. the window is the "boxcar" function). To verify this numerically, it is necessary to use a sufficiently high value for the `nfreqs` argument of `firwin2`.

Here's an example:

```
>>> bands = np.array([0, 0.5, 0.5, 0.6, 0.6, 1])
>>> desired = np.array([1, 1, 1, 0.5, 0.5, 0])
>>> numtaps = 33
>>> taps_ls = firls(numtaps, bands, desired)
>>> freqs = bands[[0, 1, 3, 5]]
>>> gains = desired[[0, 1, 3, 5]]
>>> taps_win = firwin2(numtaps, freqs, gains,
...                    nfreqs=8193, window=None)
>>> np.allclose(taps_ls, taps_win)
True
```

In general, the window method cannot be used as a replacement for the least squares method, because it does not provide an option for weighting distinct bands differently; in particular, it does not allow for "don't care" frequency intervals (i.e. intervals with weight 0).

FIR filter design: Parks-McClellan

The Parks-McClellan algorithm [PM] is based on the Remez exchange algorithm [RemezAlg]. This is a "minimax" optimization; that is, it miminizes the maximum value of $|E(\omega)|$ over $0 \le \omega \le \pi$, where $E(\omega)$ is the (weighted) deviation of the actual frequency response from the desired frequency response:

$$E(\omega) = W(\omega)(A(\omega) - D(\omega)), \quad 0 \le \omega \le \pi, \tag{6}$$

We won't give a detailed description of the algorithm here; most texts on digital signal processing explain the algorithm (e.g. Section 7.7 of Oppenheim and Schafer [OS]). The method is implemented in `scipy.signal` by the function `remez`.

As an example, we'll design a bandpass filter for a signal with a sampling rate of 2000 Hz using `remez`. For this filter, we want the stop bands to be [0, 250] and [700, 1000], and the pass band to be [350, 550]. We'll leave the behavior outside these bands unspecified, and see what `remez` gives us. We'll use 31 taps.

```
fs = 2000
bands = [0, 250, 350, 550, 700, 0.5*fs]
desired = [0, 1, 0]

numtaps = 31

taps = remez(numtaps, bands, desired, fs=fs)
```

The frequency response of this filter is the curve labeled `(a)` in Fig. 14.

To reduce the ripple in the pass band while using the same filter length, we'll adjust the weights, as follows:

```
weights = [1, 25, 1]
taps2 = remez(numtaps, bands, desired, weights, fs=fs)
```

The frequency response of this filter is the curve labeled `(b)` in Fig. 14.

It is recommended to always check the frequency response of a filter designed with `remez`. Figure 15 shows the frequency response of the filters when the number of taps is increased from 31 to 47. The ripple in the pass and stop bands is decreased, as expected, but the behavior of the filter in the interval [550, 700] might be unacceptable. This type of behavior is not unusual for filters designed with `remez` when there are intervals with unspecified desired behavior.

In some cases, the exchange algorithm implemented in `remez` can fail to converge. Failure is more likely when the number of taps is large (i.e. greater than 1000). It can also happen that `remez` converges, but the result does not have the expected equiripple behavior in each band. When a problem occurs, one can try increasing the `maxiter` argument, to allow the algorithm more iterations before it gives up, and one can try increasing `grid_density` to increase the resolution of the grid on which the algorithm seeks the maximum of the response errors.

FIR filter design: linear programming

The design problem solved by the Parks-McClellan method can also be formulated as a linear programming problem ([Rabiner1972a], [Rabiner1972b]).

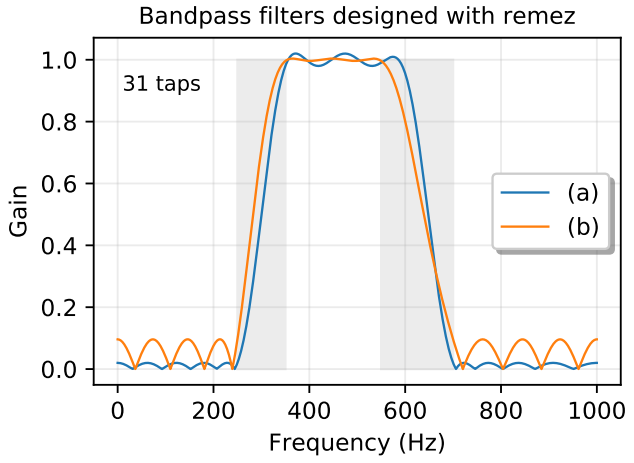

Fig. 14: *Frequency response of bandpass filters designed using* `scipy.signal.remez`*. The stop bands are [0, 250] and [700, 1000], and the pass band is [350, 550]. The shaded regions are the "don't care" intervals where the desired behavior of the filter is unspecified. The curve labeled (a) uses the default weights--each band is given the same weight. For the curve labeled (b), weight = [1, 25, 1] was used.*

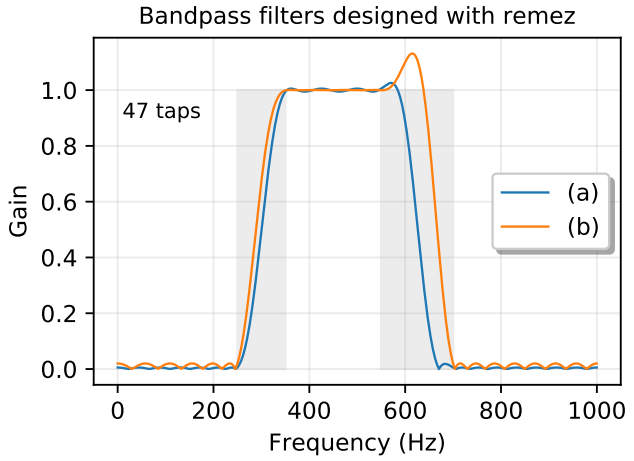

Fig. 15: *This plot shows the results of the same calculation that produced Figure 14, but the number of taps has been increased from 31 to 47. Note the possibly undesirable behavior of the filter in the transition interval [550, 700].*

To implement this method, we'll use the function `linprog` from `scipy.optimize`. In particular, we'll use the interior point method that was added in SciPy 1.0. In the following, we first review the linear programming formulation, and then we discuss the implementation.

Formulating the design problem as a linear program. Like the Parks-McClellan method, this approach is a "minimax" optimization of Eq. (6). We'll give the formulation for a Type I filter design (that is, an odd number of taps with even symmetry), but the same ideas can be applied to other FIR filter types.

For convenience, we'll consider the FIR filter coefficients for a filter of length $2R+1$ using *centered* indexing:

$$b_{-R}, b_{-R+1}, \ldots, b_{-1}, b_0, b_1, \ldots, b_{M-1}, b_R$$

Consider a sinusoidal signal with frequency ω radians per sample. The frequency response can be written

$$A(\omega) = \sum_{i=-R}^{R} b_i \cos(\omega i) = b_0 + \sum_{i=0}^{R} 2b_i \cos(\omega i) = \sum_{i=0}^{R} p_i \cos(\omega i)$$

where we define $p_0 = b_0$ and, for $1 \le i \le R$, $p_i = 2b_i$. We've used the even symmetry of the cosine function and the of filter coefficients about the middle coefficient ($b_{-i} = b_i$).

The "minimax" problem is to minimize the maximum error. That is, choose the filter coefficients such that

$$|E(\omega)| \le \varepsilon \quad \text{for} \quad 0 \le \omega \le \pi$$

for the smallest possible value of ε. After substituting the expression for $E(\omega)$ from Eqn. (6), replacing the absolute value with two inequalities, and doing a little algebra, the problem can be written as

$$\begin{aligned} \text{minimize} \quad & \varepsilon \\ \text{over} \quad & \{p_0, p_1, \ldots, p_R, \varepsilon\} \\ \text{subject to} \quad & A(\omega) - \frac{\varepsilon}{W(\omega)} \le D(\omega) \\ \text{and} \quad & -A(\omega) - \frac{\varepsilon}{W(\omega)} \le -D(\omega) \end{aligned}$$

ω is a continuous variable in the above formulation. To implement this as a linear programming problem, we use a suitably dense grid of L frequencies $\omega_0, \omega_1, \ldots, \omega_{L-1}$ (not necessarily uniformly spaced). We define the $L \times (R+1)$ matrix C as

$$C_{ij} = \cos(\omega_{i-1}(j-1)), \quad 1 \le i \le L \text{ and } 1 \le j \le R+1 \qquad (7)$$

Then the vector of frequency responses is the matrix product $C\mathbf{p}$, where $\mathbf{p} = [p_0, p_1, \ldots, p_R]^\mathsf{T}$.

Let $d_k = D(\omega_k)$, and $\mathbf{d} = [d_0, d_1, \ldots, d_{L-1}]^\mathsf{T}$. Similarly, define $\mathbf{v} = [v_0, v_1, \ldots, v_{L-1}]^\mathsf{T}$, where $v_k = 1/W(\omega_k)$. The linear programming problem is

$$\begin{aligned} \text{minimize} \quad & \varepsilon \\ \text{over} \quad & \{p_0, p_1, \ldots, p_R, \varepsilon\} \\ \text{subject to} \quad & \begin{bmatrix} C & -\mathbf{v} \\ -C & -\mathbf{v} \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ \varepsilon \end{bmatrix} \le \begin{bmatrix} \mathbf{d} \\ -\mathbf{d} \end{bmatrix} \end{aligned}$$

This is the formulation that can be used with, for example, `scipy.optimize.linprog`.

This formulation, however, provides no advantages over the solver provided by `remez`, and in fact it is generally much slower and less robust than `remez`. When designing a filter beyond a hundred or so taps, there is much more likely to be a convergence error in the linear programming method than in `remez`.

The advantage of the linear programming method is its ability to easily handle additional constraints. Any constraint, either equality or inequality, that can be written as a linear constraint can be added to the problem.

We will demonstrate how to implement a lowpass filter design using linear programming with the constraint that the gain for a constant input is exactly 1. That is,

$$A(0) = \sum_{i=0}^{R} p_i = 1$$

which may be written

$$A_{\text{eq}} \begin{bmatrix} \mathbf{p} \\ \varepsilon \end{bmatrix} = 1,$$

where $A_{\text{eq}} = [1, 1, \ldots, 1, 0]$.

Implementing the linear program. Let's look at the code required to set up a call to `linprog` to design a lowpass filter with a pass band of $[0, \omega_p]$ and a stop band of $[\omega_s, \pi]$, where the frequencies ω_p and ω_s are expressed in radians per sample, and $0 < \omega_p < \omega_s < \pi$. We'll also impose the constraint that $A(0) = 1$.

A choice for the density of the frequency samples on $[0, \pi]$ that works well is $16N$, where N is the number of taps (`numtaps` in the code). Then the number of samples in the pass band and the stop band can be computed as

```
density = 16*numtaps/np.pi
numfreqs_pass = int(np.ceil(wp*density))
numfreqs_stop = int(np.ceil((np.pi - ws)*density))
```

The grids of frequencies on the pass and stop bands are then

```
wpgrid = np.linspace(0, wp, numfreqs_pass)
wsgrid = np.linspace(ws, np.pi, numfreqs_stop)
```

We will impose an equality constraint on $A(0)$, so we can can remove that frequency from `wpgrid`--there is no point in requiring both the equality and inequality constraints at $\omega = 0$. Then `wpgrid` and `wsgrid` are concatenated to form `wgrid`, the grid of all the frequency samples.

```
wpgrid = wpgrid[1:]
wgrid = np.concatenate((wpgrid, wsgrid))
```

Let `wtpass` and `wtstop` be the constant weights that we will use in the pass and stop bands, respectively. We create the array of weights on the grid with

```
weights = np.concatenate(
    (np.full_like(wpgrid, fill_value=wtpass),
     np.full_like(wsgrid, fill_value=wtstop)))
```

The desired values of the frequency response are 1 in the pass band and 0 in the stop band. Evaluated on the grid, we have

```
desired = np.concatenate((np.ones_like(wpgrid),
                          np.zeros_like(wsgrid)))
```

Now we implement Eq. (7) and create the $L \times (R+1)$ array of coefficients C that are used to compute the frequency response, where $R = M/2$:

```
R = (numtaps - 1)//2
C = np.cos(wgrid[:, np.newaxis]*np.arange(R+1))
```

The column vector of the reciprocals of the weights is

```
V = 1/weights[:, np.newaxis]
```

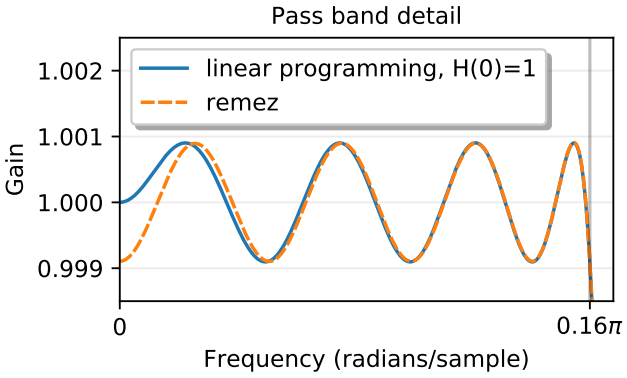

Fig. 16: *Result of solving a lowpass FIR filter design problem by linear programming with the constraint $A(0) = 1$. The response without the extra constraint, solved using `remez`, is also plotted.*

Next we assemble the pieces that define the inequality constraints that are actually passed to `linprog`:

```
A = np.block([[ C, -V],
              [-C, -V]])
b = np.block([[desired, -desired]]).T
c = np.zeros(M+2)
c[-1] = 1
```

In code, the arrays for the equality constraint needed to define $A(0) = 1$ are:

```
A_eq = np.ones((1, R+2))
A_eq[:, -1] = 0
b_eq = np.array([1])
```

Finally, we set up and call `linprog`:

```
options = dict(maxiter=5000, tol=1e-6)
sol = linprog(c, A, b, A_eq=A_eq, b_eq=b_eq,
              bounds=(None, None),
              method='interior-point',
              options=options)
if sol.success:
    p = sol.x[:-1]
    taps = 0.5*np.concatenate((p[:0:-1],
                               [2*p[0]],
                               p[1:]))
```

Notes:

- For different problems, the parameters defined in the dictionary `options` may have to be adjusted. See the documentation for `linprog` for more details.
- By default, `linprog` assumes that all the variables must be nonnegative. We use the `bounds` argument to override that behavior.
- We have had more success using the interior point method than the default simplex method.

See Figure 16 for a plot of the pass band response of the filter designed using `linprog`. The number of taps was $N = 81$, and the transition boundary frequencies, expressed in radians per sample, were $\omega_p = 0.16\pi$ and $\omega_s = 0.24\pi$. For the weight in each band we used `wtpass = 2` and `wtstop = 1`.

Determining the order of a FIR filter

Most of the filter design tools in SciPy require the number of taps as an input. Typically, however, a designer has requirements on the pass band ripple and the stop band rejection, and wants the FIR filter with the minimum number of taps that satisfies these requirements. The diagram shown in Figure 17 illustrates the design parameters for a lowpass filter. The graph of the magnitude of the frequency response of the filter must not enter the shaded area. The parameter δ_p defines the allowed pass band ripple, and δ_s defines the required attenuation in the stop band. The maximum width of the transition from the pass band to stop band is $\Delta\omega$, and the cutoff frequency ω_c is centered in the transition band.

In the next two sections, we'll consider the following filter design problem. We need a lowpass filter for a signal that is sampled at 1000 Hz. The desired cutoff frequency is 180 Hz, and the transition from the pass band to the stop band must not exceed 30 Hz. In the pass band, the gain of the filter should deviate from 1 by no more than 0.005 (i.e. worst case ripple is 0.5%). In the stop band, the gain must be less than 0.002 (about 54 dB attenuation). In the next section, we'll tackle the design using the Kaiser window method. After that, we'll obtain an optimal design by using the Parks-McClellan method.

Kaiser's window method

The Kaiser window is a window function with a parameter β that controls the shape of the function. An example of a Kaiser window is plotted in Figure 11. Kaiser [Kaiser66], [Kaiser74] developed formulas that, for a given transition width $\Delta\omega$ and error tolerance for the frequency response, determine the order M and the parameter β required to meet the requirements. Summaries of the method can be found in many sources, including Sections 7.5.3 and 7.6 of the text by Oppenheim and Schafer [OS].

In Kaiser's method, there is only one parameter that controls the passband ripple and the stopband rejection. That is, Kaiser's method assumes $\delta_p = \delta_s$. Let δ be that common value. The stop band rejection in dB is $-20\log_{10}(\delta)$. This value (in dB) is the first argument of the function `kaiserord`. One can interpret the argument `ripple` as the maximum deviation (expressed in dB) allowed in $|A(\omega) - D(\omega)|$, where $A(\omega)$ is the magnitude of the actual frequency response of the filter and $D(\omega)$ is the desired frequency response. (That is, in the pass band, $D(\omega) = 1$, and in the stop band, $D(\omega) = 0$.) In Figure 18, the bottom plot shows $|A(\omega) - D(\omega)|$.

The Kaiser window design method, then, is to determine the length of the filter and the Kaiser window parameter β using Kaiser's formula (implemented in `scipy.signal.kaiserord`), and then design the filter using the window method with a Kaiser window (using, for example, `scipy.signal.firwin`):

```
numtaps, beta = kaiserord(ripple, width)
taps = firwin(numtaps, cutoff,
              window=('kaiser', beta),
              [other args as needed])
```

For our lowpass filter design problem, we first define the input parameters:

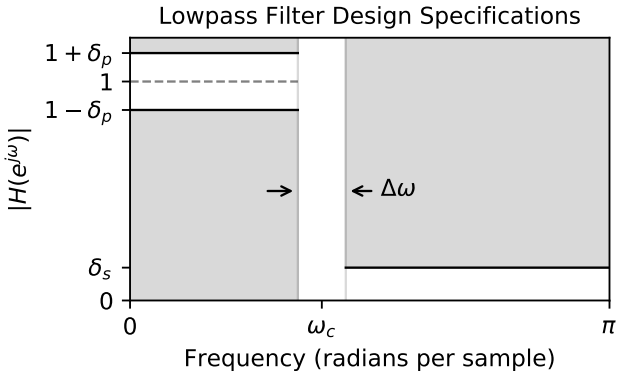

Fig. 17: *Lowpass filter design specifications. The magnitude of the frequency response of the filter should not enter the shaded regions.*

```
# Frequency values in Hz
fs = 1000.0
cutoff = 180.0
width = 30.0
# Desired pass band ripple and stop band attenuation
deltap = 0.005
deltas = 0.002
```

As already mentioned, the Kaiser method allows for only a single parameter to constrain the approximation error. To ensure we meet the design criteria in the pass and stop bands, we take the minimum of δ_p and δ_s:

```
delta = min(deltap, deltas)
```

The first argument of `kaiserord` must be expressed in dB, so we set:

```
delta_db = -20*np.log10(delta)
```

Then we call `kaiserord` to determine the number of taps and the Kaiser window parameter β:

```
numtaps, beta = kaiserord(delta_db, width/(0.5*fs))
numtaps |= 1  # Must be odd for a Type I FIR filter.
```

For our lowpass filter design problem, we find `numtaps` is 109 and β is 4.990.

Finally, we use `firwin` to compute the filter coefficients:

```
taps = firwin(numtaps, cutoff/(0.5*fs),
              window=('kaiser', beta), scale=False)
```

The results of the Kaiser method applied to our lowpass filter design problem are plotted in Figure 18. The tip of the rightmost ripple in the pass band violates the δ-constraint by a very small amount; this is not unusual for the Kaiser method. In this case, it is not a problem, because the original requirement for the pass band is $\delta_p = 0.005$, so the behavior in the pass band is overly conservative.

Optimizing the FIR filter order

The Kaiser window method can be used to create *a* filter that meets (or at least is very close to meeting) the design requirements, but it will not be optimal. That is, generally there will exist FIR filters with fewer taps that also satisfy the design requirements. At the time this chapter is being written, SciPy does not provide a tool that automatically determines

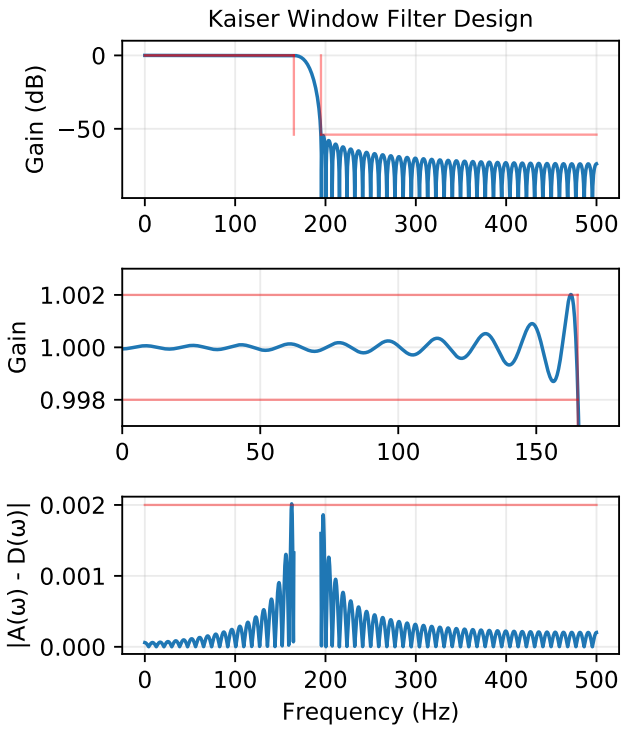

Fig. 18: *Result of the Kaiser window filter design of a lowpass filter. The number of taps is 109.* Top: *Magnitude (in dB) of the frequency response.* Middle: *Detail of the frequency response in the pass band.* Bottom: *The deviation of the actual magnitude of the frequency response from that of the ideal lowpass filter.*

the optimal number of taps given pass band ripple and stop band rejection requirements. It is not difficult, however, to use the existing tools to find an optimal filter in a few steps (at least if the filter order is not too large).

Here we show a method that works well, at least for the basic lowpass, highpass, bandpass and bandstop filters on which it has been tested. The idea: given the design requirements, first estimate the length of the filter. Create a filter of that length using `remez`, with $1/\delta_p$ and $1/\delta_s$ as the weights for the pass and stop bands, respectively. Check the frequency response of the filter. If the initial estimate of the length was good, the filter should be close to satisfying the design requirements. Based on the observed frequency response, adjust the number of taps, then create a new filter and reevaluate the frequency response. Iterate until the shortest filter that meets the design requirements is found. For moderate sized filters (up to 1000 or so taps), this simple iterative process can be automated. (For higher order filters, this method has at least two weaknesses: it might be difficult to get a reasonably accurate estimate of the filter length, and it is more likely that `remez` will fail to converge.)

A useful formula for estimating the length of a FIR filter was given by Bellanger [Bellanger]:

$$N \approx -\frac{2}{3}\log_{10}(10\delta_p\delta_s)\frac{f_s}{\Delta f} \tag{8}$$

which has a straightforward Python implementation:

```
def bellanger_estimate(deltap, deltas, width, fs):
    n = (-2/3)*np.log10(10*deltap*deltas)*fs/width
    n = int(np.ceil(n))
    return n
```

We'll apply this method to the lowpass filter design problem that was described in the previous section. As before, we define the input parameters:

```
# Frequency values in Hz
fs = 1000.0
cutoff = 180.0
width = 30.0
# Desired pass band ripple and stop band attenuation
deltap = 0.005
deltas = 0.002
```

Then the code

```
numtaps = bellanger_estimate(deltap, deltas,
                             width, fs)
numtaps |= 1
```

gives `numtaps = 89`. (Compare this to the result of the Kaiser method, where `numtaps` is 109.)

Now we'll use `remez` to design the filter.

```
trans_lo = cutoff - 0.5*width
trans_hi = cutoff + 0.5*width
taps = remez(numtaps,
             bands=[0, trans_lo,
                    trans_hi, 0.5*fs],
             desired=[1, 0],
             weight=[1/deltap, 1/deltas],
             fs=fs)
```

The frequency response of the filter is shown in Figure 19. We see that the filter meets the design specifications. If we decrease the number of taps to 87 and check the response, we find that the design specifications are no longer met, so we accept 89 taps as the optimum.

References

[Bellanger] M. Bellanger, *Digital Processing of Signals: Theory and Practice* (3rd Edition), Wiley, Hoboken, NJ, 2000.

[Kaiser66] J. F. Kaiser, Digital filters, in *System Analysis by Digital Computer*, Chapter 7, F. F. Kuo and J. F. Kaiser, eds., Wiley, New York, NY, 1966

[Kaiser74] J. F. Kaiser, Nonrecursive digital filter design using the I0-sinh window function, *Proc. 1974 IEEE International Symp. on Circuits and Systems*, San Francisco, CA, 1974.

[Lyons] Richard G. Lyons. *Understanding Digital Signal Processing* (2nd ed.), Pearson Higher Education, Inc., Upper Saddle River, New Jersey (2004)

[OS] Alan V. Oppenheim, Ronald W. Schafer. *Discrete-Time Signal Processing* (3rd ed.), Pearson Higher Education, Inc., Upper Saddle River, New Jersey (2010)

[PM] Parks-McClellan filter design algorithm. Wikipedia, https://en.wikipedia.org/wiki/Parks%E2%80%93McClellan_filter_design_algorithm

[Rabiner1972a] L. R. Rabiner, The design of finite impulse response digital filters using linear programming techniques, *The Bell System Technical Journal*, Vol. 51, No. 6, July-August, 1972.

[Rabiner1972b] L. R. Rabiner, Linear program design of finite impulse response (FIR) digital filters, *IEEE Trans. on Audio and Electroacoustics*, Vol. AU-20, No. 4, Oct. 1972.

[RemezAlg] Remez algorithm. Wikipedia, `https://en.wikipedia.org/wiki/Remez_algorithm`

[SavGol] A. Savitzky, M. J. E. Golay. Smoothing and Differentiation of Data by Simplified Least Squares Procedures. *Analytical Chemistry*, 1964, 36 (8), pp 1627-1639.

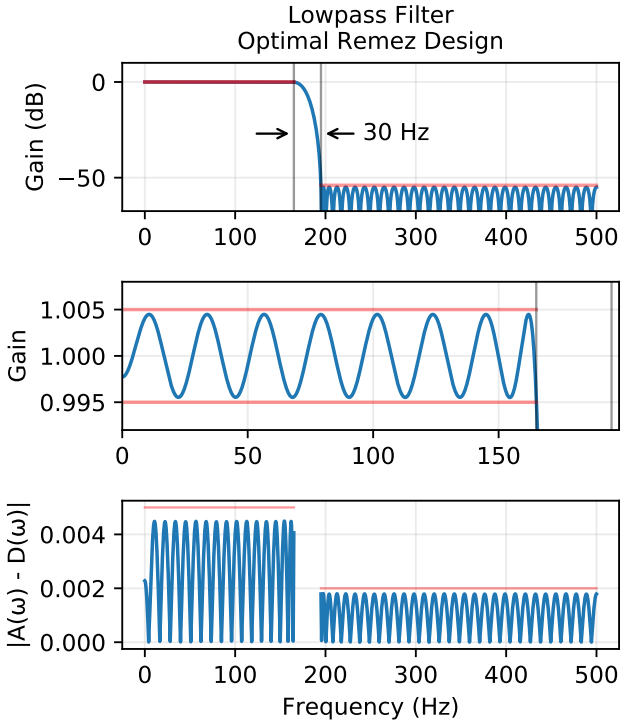

Fig. 19: *Optimal lowpass filter frequency response. The number of taps is 89.*

[SO] Nimal Naser, How to filter/smooth with SciPy/Numpy?, https://stackoverflow.com/questions/28536191