AI Accelerated Digital Filter Design: Butterworth, Chebyshev, Elliptic, and General IIR Filters

Julius Smith
CCRMA, Stanford University

Music 320 Extensions - Digital Filter Design

Outline

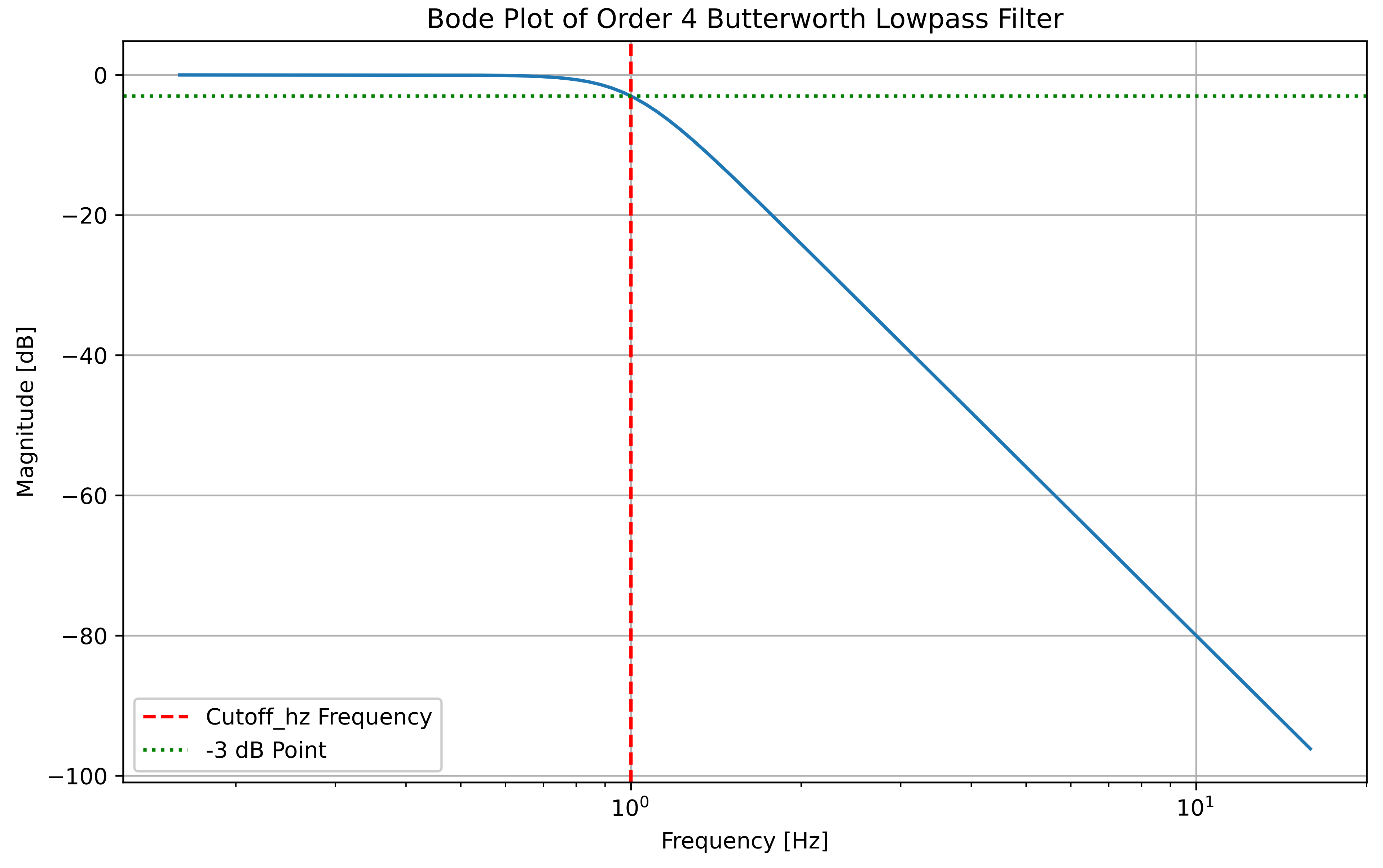
- Example Classic Analog Filters (Butterworth, Chebyshev, Elliptic)
- Digitizing Analog Filters (two ways)
- Relating s and z planes
- Classic Analog Filter Design
  - Butterworth (maximally flat passband, smooth rolloff)
  - Chebyshev (equiripple passband, Butterworth stopband [or vice versa])
  - Elliptic (equiripple passband and stopband)
- Butterworth Filters in Python, Faust, and C++
- General Digital Filter Design (not starting from Analog)

AI was used *throughout* for

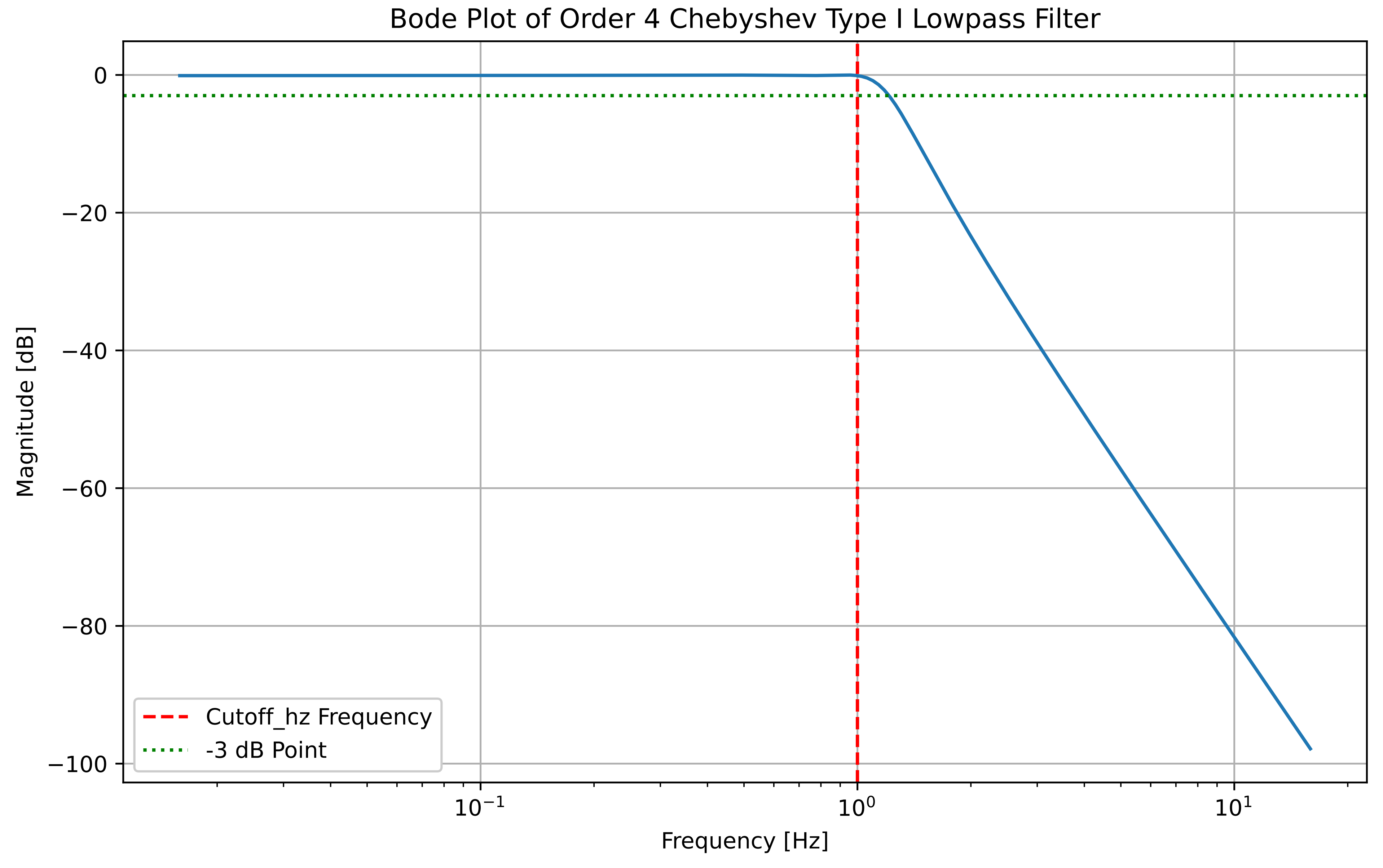
- LaTeX typesetting
- Python code for all figures
- Python and C++ functions for filter design (not in `scipy.signal`)
- In general, Claude 3.5 Sonnett was used (often in Cursor or VS Code)

Classic Analog Lowpass Filters

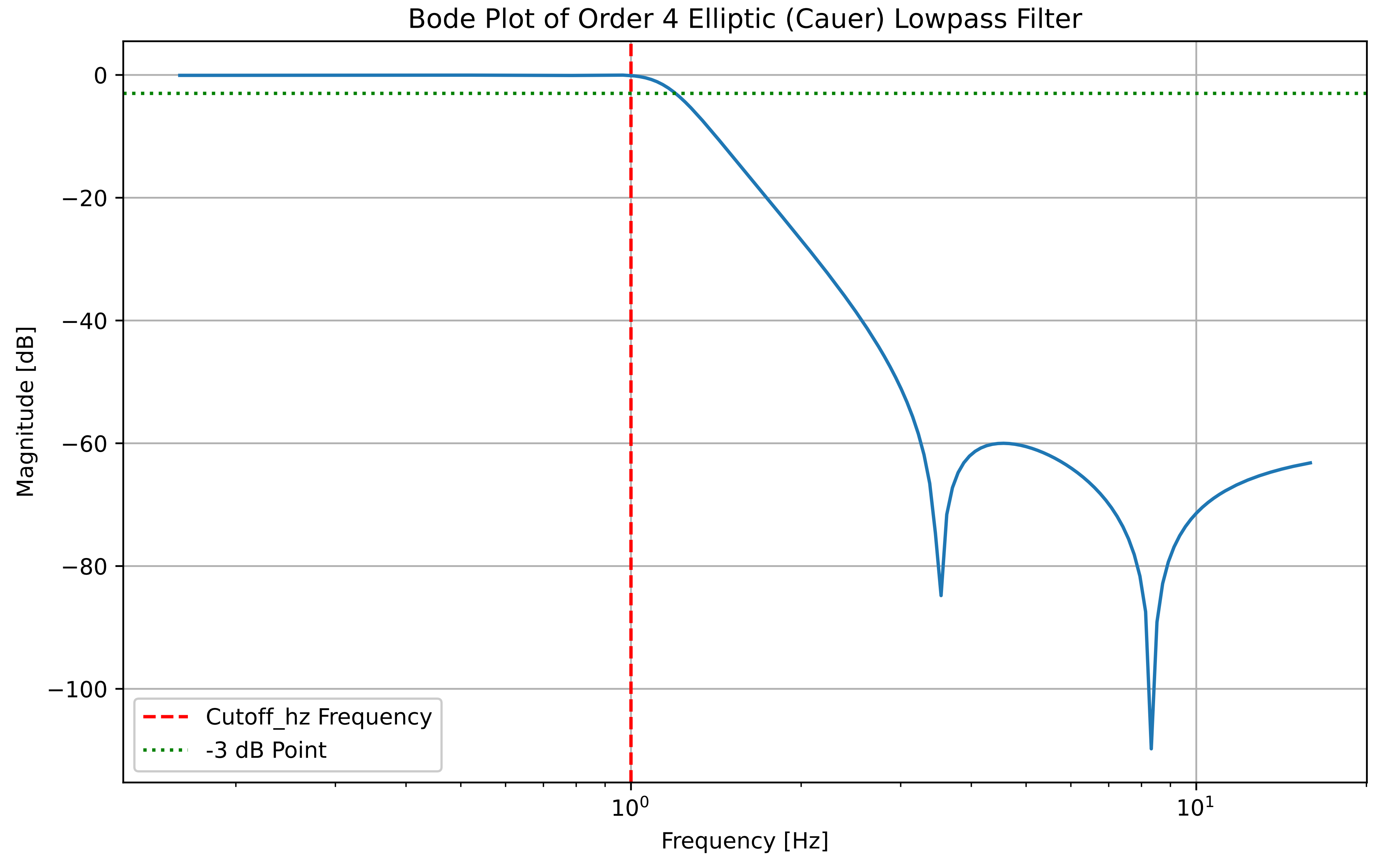
Butterworth Analog Lowpass Prototype Example



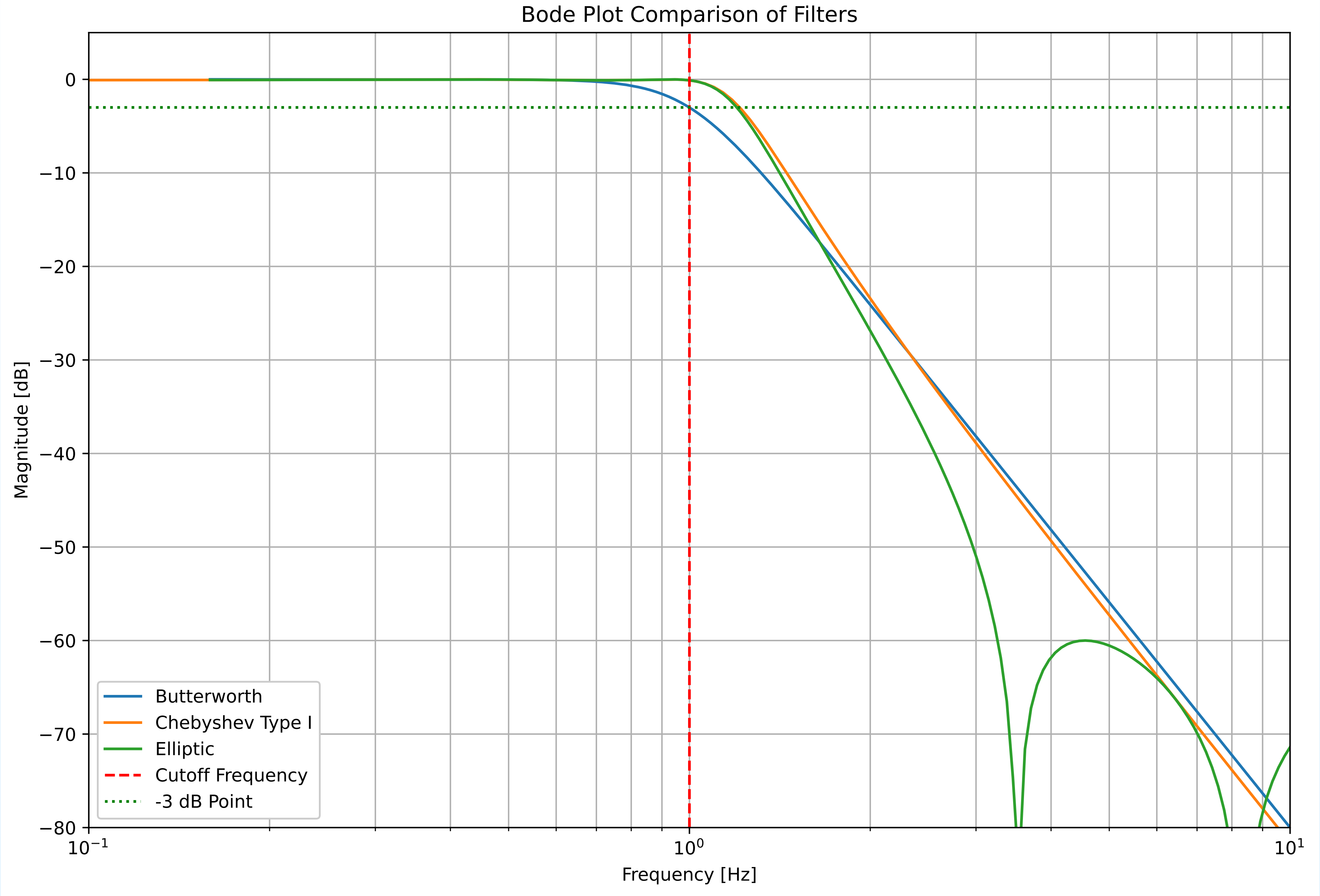
Chebyshev1 Analog Lowpass Prototype Example



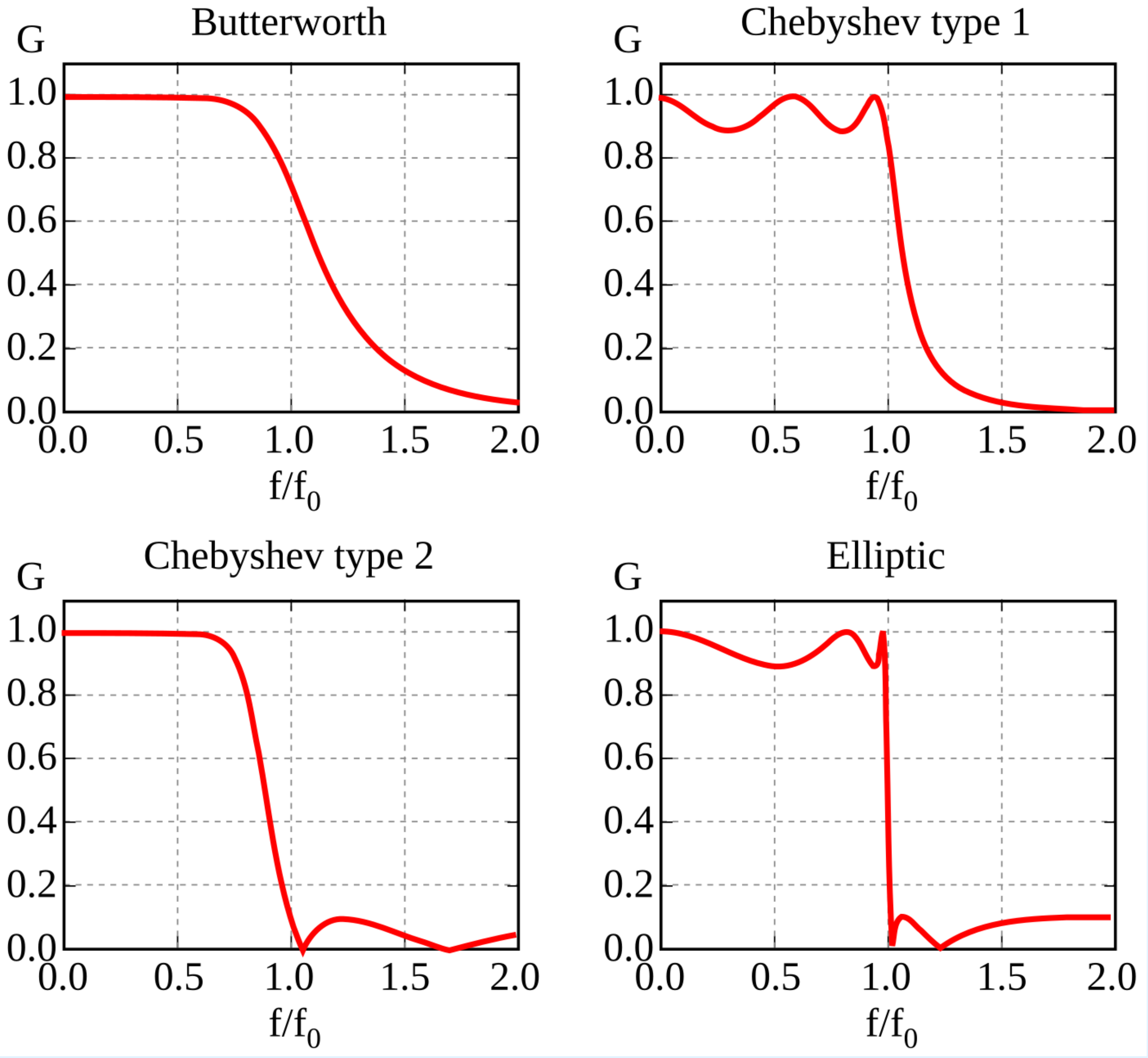
Elliptic Analog Lowpass Prototype Example



Butterworth, Chebyshev I, and Elliptic Analog Lowpasses Overlaid



Butterworth, Chebyshev I *and* II, Elliptic Analog Lowpasses, Wikipedia



Python Main Program Used Above (by Claude 3.5 Sonnet)

```python
order = 4  # Filter order
cutoff_hz = 1 # Cutoff_hz frequency in Hz
ripple = 0.1  # Passband ripple in dB
sb_att = 60  # Stopband attenuation (ripple) in dB

wB, HB = butterworth_lowpass(order, cutoff_hz)
title = f'Order {order} Butterworth Lowpass'
plot_bode(wB, HB, title, save_path="...")

wC1, HC1 = chebyshev1_lowpass(order, cutoff_hz, ripple)
title = f'Order {order} Chebyshev Type I Lowpass'
plot_bode(wC1, HC1, title, save_path="...")

wE, HE = elliptic_lowpass(order, cutoff_hz, ripple, sb_att)
title = f'Order {order} Elliptic (Cauer) Lowpass'
plot_bode(wE, HE, title, save_path="...")


...
```

Python for Filter Design (by Claude 3.5 Sonnet)

```
def butterworth_lowpass(order, cutoff_hz):
    w, H = signal.freqs(*signal.butter(order, cutoff_hz * 2 * np.pi, btype='lowpass', analog=True))
    return w, H
```

```
def chebyshev1_lowpass(order, cutoff, ripple):
    w, H = signal.freqs(*signal.cheby1(order, ripple, cutoff_hz * 2 * np.pi, btype='lowpass', analog=True))
    return w, H
```

```
def elliptic_lowpass(order, cutoff, ripple, stopband_attenuation):
    w, H = signal.freqs(*signal.ellip(order, ripple, stopband_attenuation, cutoff_hz * 2 * np.pi, btype='lowpass', analog=True))
    return w, H
```

The s and z Planes

Generalized sinusoids in continuous and discrete time:

Continuous Time

$$
\begin{aligned}
e^{st} &= e^{(\sigma+j\omega)t} \\
&= e^{\sigma t} e^{j\omega t} \\
&= e^{-t/\tau}\left[\cos(\omega t) + j\sin(\omega t)\right]
\end{aligned}
$$

Discrete Time when $z = e^{sT}$

$$
\begin{aligned}
z^n &= \left(e^{sT}\right)^n = \left(e^{\sigma T + j\omega T}\right)^n \\
&= e^{\sigma nT} e^{j\omega nT} \\
&= e^{-nT/\tau}\left[\cos(\omega\, nT) + j\sin(\omega\, nT)\right]
\end{aligned}
$$

Laplace Transform

$$X_c(s) = \int_0^\infty x_c(t) e^{-st} dt$$

z Transform

$$X_d(z) = \sum_{n=0}^{\infty} x_d(n) z^{-n}$$

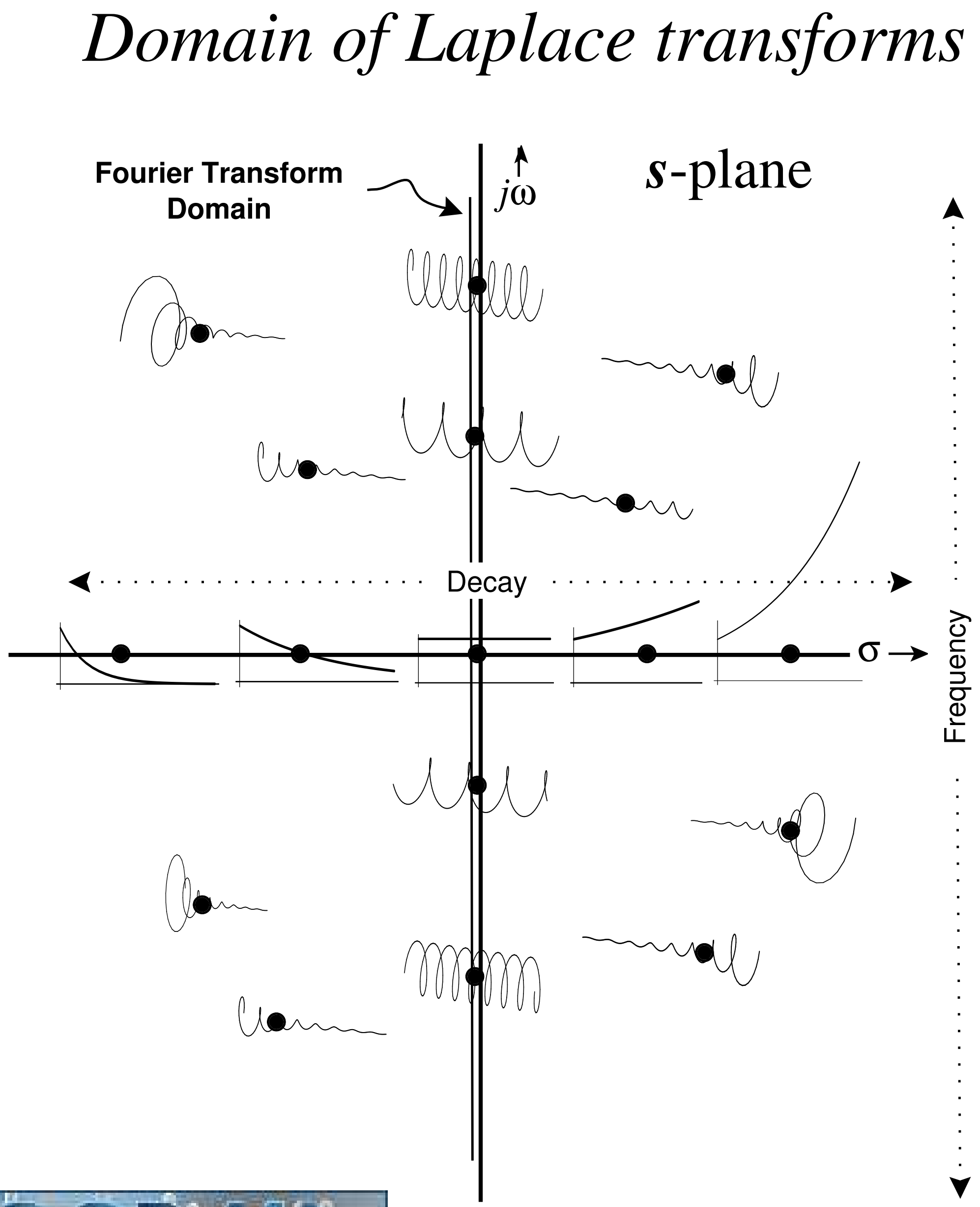
Fourier Transform (FT) ($s = j\omega$)

$$X_c(j\omega) = \int_0^\infty x_c(t) e^{-j\omega t} dt$$

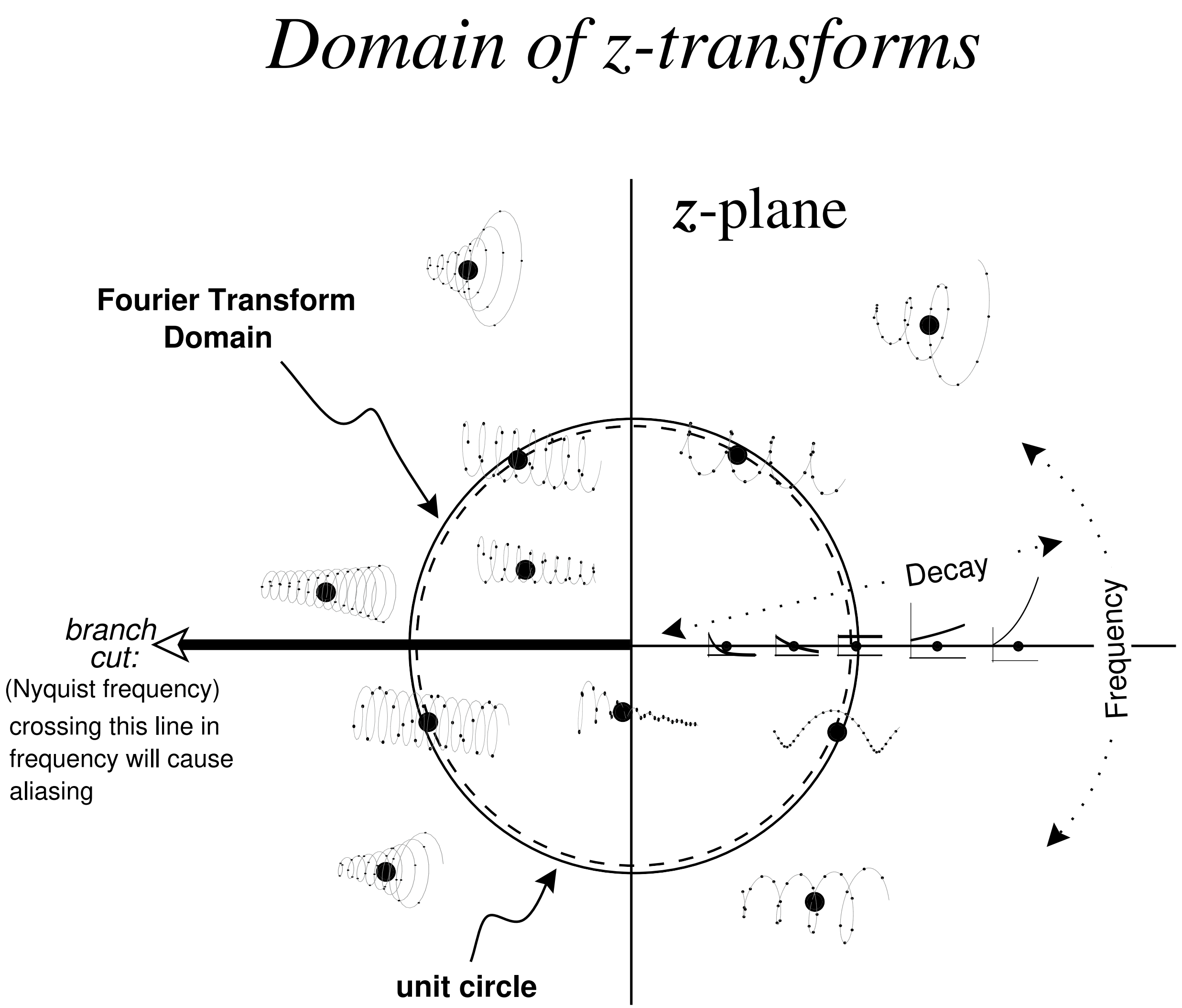
Discrete Time FT (DTFT) ($z = e^{j\omega T}$)

$$X_d(e^{j\omega T}) = \sum_{n=0}^{\infty} x_d(n) e^{-j\omega T}$$

Generalized Sinusoids e^{st} in the s Plane



Generalized Sinusoids z^n in the z Plane



Impulse Invariance Filter Digitization

The Impulse Invariance Method for digitizing analog filters effectively *samples* their *impulse response*.
An analog transfer function is always a *rational* function of s:

$$H(s) = \frac{B(s)}{A(s)}$$

- $B(s) =$ numerator polynomial having roots called the *zeros* of $H(s)$
- $A(s) =$ denominator polynomial having roots called the *poles* of $H(s)$.

In *partial fractions,*

$$H(s) = \sum_{i=1}^{N} \frac{K_i}{s+s_i}. \tag{1}$$

The impulse response $h(t)$ is the *inverse Laplace transform* of (1):

$$h(t) = \sum_{i=1}^{N} K_i e^{-s_i t}$$

Impulse Invariance Method, Continued

We have

$$h(t) = \sum_{i=1}^{N} K_i e^{-s_i t}$$

Let's now *sample* $h(t)$ at intervals of T seconds:

$$h(nT) = \sum_{i=1}^{N} K_i e^{-s_i nT} = \sum_{i=1}^{N} K_i \left(e^{-s_i T}\right)^n \triangleq \sum_{i=1}^{N} K_i z_i^n$$

We see that each *analog pole* at $s = -s_i$ maps to a *digital pole* at

$$\boxed{z_i = e^{-s_i T}.}$$

While the *analog zeros* do not map in a simple way to the z plane, the *pole residues* K_i are preserved unchanged.

Bilinear Transform

An alternative to *sampling* in the time-domain for *systems* (as opposed to signals) is to start in the *frequency domain* and apply the *Bilinear Transform:*

$$\boxed{s = \alpha \frac{1 - z^{-1}}{1 + z^{-1}}} \qquad z^{-1} = \frac{1 - s/\alpha}{1 + s/\alpha}$$

- α is any positive constant
- Setting $\alpha = 2/T$ matches *low frequencies* relative to the sampling rate f_s
- More generally, α can map *any one frequency* exactly
- See also Cayley (1846) and Möbius transforms
- Can show:
  - Analog frequency axis $s = j\omega$ (vertical axis in the s plane) maps exactly *once* to the digital frequency axis $z = e^{j\omega T}$ (unit circle in the z plane) $\Rightarrow$ *no aliasing*
  - The *left half* of the s plane (stability region for *poles*) maps to the *interior* of the unit circle in the z plane (its stability region) $\Rightarrow$ *stability preserved*

Oversampling Gives $\mathbf{z \approx 1 + sT}$

At low frequencies and dampings, *i.e.*, near $s \approx 0$ and $z \approx 1$, we have the following low-frequency approximations (low relative to the sampling rate):

- **Basic Sampling:**

$$z \;=\; e^{sT} \;=\; 1 + sT + \frac{(sT)^2}{2!} + \frac{(sT)^3}{3!} + \cdots \;\approx\; \boxed{1 + sT}$$

- **Bilinear Transform:**

$$z \;=\; \frac{1 + s/\alpha}{1 - s/\alpha} \;=\; \left(1 + \frac{s}{\alpha}\right)\left[1 + \frac{s}{\alpha} + \left(\frac{s}{\alpha}\right)^2 + \cdots\right] \;\approx\; 1 + 2\frac{s}{\alpha} \;=\; \boxed{1 + sT}$$

when $\alpha = 2/T$

It is good to oversample sufficiently so that there is no audible difference between the z-planes of signals and systems digitized separately by ordinary sampling and the bilinear transform (or even *multiple* bilinear transforms, as in Wave Digital Filters).

Classic Analog Filters Derived

IIR Digital Filter Design

Digitizing **Analog Prototypes** (Lowpass, Highpass, Bandpass) typically starts with

- Butterworth
  - Maximally flat passband
  - Poles on a *circle* in s and z planes
  - All zeros at *infinity* in the s plane
  - All zeros at $z = -1$ in the z plane
- Chebyshev Type I
  - Equiripple passband ("Chebyshev in the Passband")
  - Poles along an *ellipse* in the s plane
  - "Butterworth in the Stopband"
    - All zeros at *infinity* in the s plane
    - All zeros at $z = -1$ in the z plane
- Chebyshev Type II
  - "Chebyshev in the Stopband"
  - "Butterworth in the Passband"
  - Zeros along *frequency axis* $s = j\omega$ or $z = e^{j\omega T}$
- Elliptic (Cauer)
  - "Chebyshev in the Stopband"
  - "Chebyshev in the Passband"
  - Poles along *ellipse*
  - Zeros along *frequency axis*
- See also MIT Open CourseWare, and
  - https://en.wikipedia.org/wiki/Butterworth_filter
  - https://en.wikipedia.org/wiki/Chebyshev_filter
  - https://en.wikipedia.org/wiki/Elliptic_filter

IIR Digital Filter Design Reference

My go-to book:

Digital Filter Design
T. W. Parks and C. S. Burrus
John Wiley and Sons, Inc., New York, 1987

The derivations below follow Parks and Burrus, using mostly the same notation.

Taylor Series Expansion

Any transfer function $F(\omega)$ can be expanded as a Taylor series:

$$F(\omega) = K_0 + K_1\,\omega + K_2\,\omega^2 + K_3\,\omega^3 + \cdots,$$

where

$$K_0 = F(0), \quad K_1 = \left.\frac{dF(\omega)}{d\omega}\right|_{\omega=0}, \quad K_2 = \frac{1}{2}\left.\frac{d^2F(\omega)}{d\omega^2}\right|_{\omega=0},$$

The *power response*

$$\mathcal{F}(j\omega) = F(\omega)\cdot\overline{F(\omega)}$$

expands as a polynomial in ω^2 with *real coefficients*:

$$\mathcal{F}(\omega) = k_0 + k_2\,\omega^2 + k_4\,\omega^4 + \cdots$$

$\mathcal{F}(\omega)$ is *maximally flat about dc* when $k_2 = k_4 = k_6 = \cdots = k_{2N} = 0$, where N is the filter order. That is, *all degrees of freedom* in the filter (other than scaling) are used to *flatten* the power response near dc (0 Hz).

Rational Function Form

An order N filter power response can also be expressed as a *rational function* of ω^2:

$$\mathcal{F}(j\omega) = \frac{d_0 + d_2\,\omega^2 + d_4\,\omega^4 + \cdots + d_{2M}\,\omega^{2M}}{c_0 + c_2\,\omega^2 + c_4\omega^4 + \cdots + c_{2N}\,\omega^{2N}},\ M \leq N \tag{1}$$

In any passband, we can define the error $E(\omega)$ as the *deviation* from a gain of 1:

$$\mathcal{F}(j\omega) = 1 + E(\omega) \tag{2}$$

Combining (1) and (2) gives:

$$d_0 + d_2\,\omega^2 + \cdots + d_{2M}\,\omega^{2M} = c_0 + c_2\,\omega^2 + \cdots + c_{2N}\,\omega^{2N} + E(\,\omega)[c_0 + c_2\,\omega^2 + \cdots] \tag{3}$$

Padé Approximation (to the constant d_0/c_0) *zeros as many leading terms as possible in the series expansion of the error*. (This also yields the *maximally flat passband*.) Thus, we set

$$\begin{aligned} c_0 &= d_0 & c_2 &= d_2 & \cdots && c_{2M} &= d_{2M} \\ c_{2M+2} &= 0 & c_{2M+4} &= 0 & \cdots && c_{2N-2} &= 0 \\ c_{2N} &= \text{nonzero} \end{aligned} \tag{4}$$

Butterworth Filters

Butterworth Power Response

The power response of the normalized Nth-order Butterworth filter is given by

$$\mathcal{F}(j\omega) = \frac{1}{1+\omega^{2N}}$$

- cutoff frequency is normalized to $\omega = 1$
- To set cutoff frequency ω_c, use $\mathcal{F}(j\omega/\omega_c)$
- *All zeros at infinity* so that filter "rolls off" $-6N$ dB/octave above cutoff
- The general case gets $d_0 = 1$, $c_0 = 1$, $c_k = 0$ for $0 < k < 2N$, and $c_{2N} = 1$
- Lowpass passband is *maximally flat*
  (Padé approximation to $\mathcal{F}(j\omega) \approx 1$ about dc)

Butterworth Transfer Function

Power response of the normalized Nth-order Butterworth filter:

$$\mathcal{F}(j\omega) = \frac{1}{1+\omega^{2N}}$$

Recall that

$$\mathcal{F}(j\omega) \triangleq F(j\omega)\,\overline{F(j\omega)} \;=\; F(s)\,F(-s)\big|_{s=j\omega}$$

Thus, in the s-domain, we have

$$F(s)F(-s) \;=\; \frac{1}{1+[(s/j)^2]^N} \;=\; \frac{1}{1+(-s^2)^N},$$

where $s \triangleq \sigma + j\omega$.

Pole Locations

We have

$$F(s)F(-s) \;=\; \frac{1}{1+(-s^2)^N}.$$

The locations of the N poles are therefore given by

$$\sigma_k = -\cos\left(\frac{k\pi}{2N}\right), \quad \omega_k = \sin\left(\frac{k\pi}{2N}\right)$$

for N values of k where

$$k = \pm 1, \pm 3, \pm 5, \ldots, \pm(N-1) \quad \text{for N even,}$$

$$k = 0, \pm 2, \pm 4, \ldots, \pm(N-1) \quad \text{for N odd.}$$

Factored Form of F(s)

Even F(s) has the partially factored form

$$F(s) = \prod_k \frac{1}{s^2 + 2\cos(k\pi/2N)s + 1}$$

for $k = 1, 3, 5, \ldots, N-1$.

For N odd, F(s) has a single real pole:

$$F(s) = \frac{1}{s+1} \prod_k \frac{1}{s^2 + 2\cos(k\pi/2N)s + 1}$$

Butterworth Poles in s and z Planes

Claude 3.5 Sonnet Prompt 1:

(Dictating by voice, then editing "airplane" to "s-plane" etc. in the prompt)

> "Write a python script that plots the poles of a Butterworth filter on the left in the s-plane and on the right in the z-plane, given the filter order as a parameter. The filter cutoff frequency is 1 rad/s, and the sampling rate for the z-plane case is another parameter between 2 and 10 rad/s."

(Result: Excellent, except that all poles were plotted in the lower half-plane.)

Prompt 2:

> "This is a good start. However, the Butterworth poles should be in the left-half plane, not the bottom-half. Also please draw the unit circle with a dashed line. All poles should lie on it."

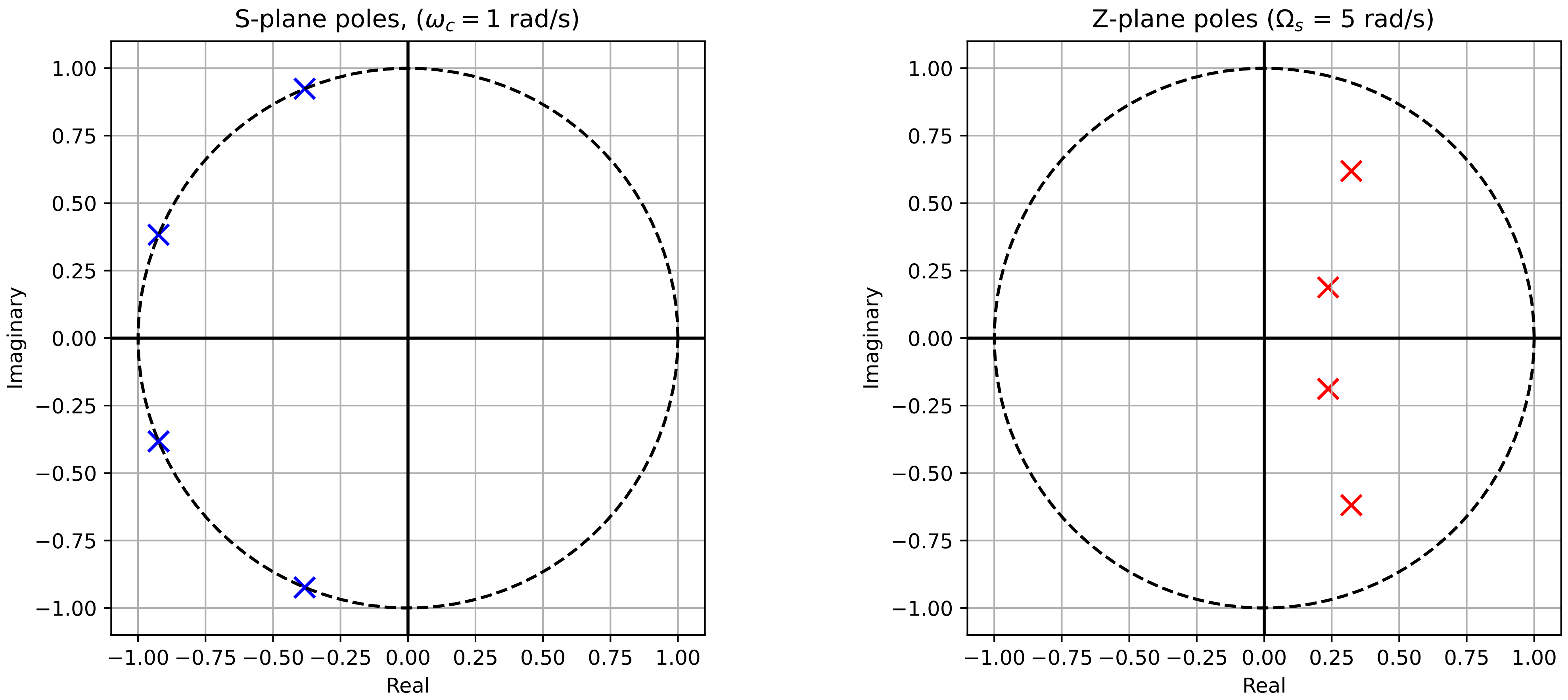
(Result: Nailed it.)

Final Tweaks:

I placed the Python file in its own directory, `cd`'d there, typed "`cursor .`", and dictated "This plot is nice, but the z-plane case on the right should mention the sampling rate in its title. Also, the plot should be saved to the file `ButterworthPoles.eps`

Butterworth Poles in s and z Planes Plotted

Result:



Butterworth Filters in FAUST

Perhaps the easiest path to Butterworth filters in C++ is via FAUST:

- `https://faustlibraries.grame.fr/libs/filters/#filowpass`
- $\cdots$

The *graphic equalizer* filter bank is also based on Butterworth band-splits:

- `https://faustlibraries.grame.fr/libs/filters/#fifilterbank`
- Arbitrary spectral partitions are supported
- Bands sum to a *constant magnitude frequency response* when all gains are 1
- *Odd-order* Butterworth band-splits are required
- Reference:
  "Tree-structured complementary filter banks using all-pass sections"
  Regalia et al., IEEE Trans. Circuits & Systems, CAS-34:1470-1484, Dec. 1987

FAUST Test Program

Try this in the `https://faustide.grame.fr`:

```
import("stdfaust.lib");
cutoff = hslider("cutoff",5000,20,10000,1);
process = ba.pulsen(1, 10000) : fi.lowpass(3,cutoff);
```

Normalized Second-Order Analog Lowpass

The transfer function of a *normalized* second-order lowpass can be written as

$$H_l(s) = \frac{1}{\tilde{s}^2 + \frac{1}{Q}\tilde{s} + 1}, \quad \tilde{s} \triangleq \frac{s}{\omega_c},$$

where the normalization $\tilde{s} = s/\omega_c$ maps the desired corner frequency ω_c to 1, and the "quality factor" Q is defined as

$$Q \triangleq \frac{\omega_c}{2\alpha}$$

where α is minus the real part of the complex-conjugate pole locations p and $\overline{p}$:

$$p, \overline{p} = -\alpha \pm j\sqrt{\omega_c^2 - \alpha^2}$$

A Word About Q

- We defined the "quality factor" Q for a second-order (potentially *resonant*) *lowpass* as

$$Q \triangleq \frac{\omega_c}{2\alpha}$$

  where ω_c is the *corner frequency* and α is minus the poles' real part.
- The Q of a resonance is normally defined as *center frequency over bandwidth*
  `https://ccrma.stanford.edu/~jos/filters/Quality_Factor_Q.html`
- A real pole at $s = -\alpha$ in fact has its -3dB points at $\omega = \pm\alpha$, giving 3dB *bandwidth* 2α radians per second (centered on $\omega = 0$)
- Shifting that pole up to $s = -\alpha + j\omega_c$ gives a *complex* resonator with bandwidth 2α
- Adding that to a pole at $s = -\alpha - j\omega_c$ gives a *real* resonator, which we can view as the *superposition* of two complex resonators, each having bandwith 2α
- In this way, 2α may be regarded as the bandwidth of the *corner resonance* at ω_c, even when the resonance is not prominent, such as for $Q \leq \sqrt{2}/2$, as we'll see:

Corner Resonance

- The normalized second-order *Butterworth* lowpass has poles on the unit circle of the s plane at $p, \overline{p} = (-1 \pm j)\sqrt{2}/2$
- Therefore, $\alpha = \sqrt{2}/2$ so that $Q = 1/\sqrt{2}$ for this case
- Larger Q values give the “corner resonance” effect often used in music synthesizers
- Let’s now plot the *magnitude frequency response* of some second-order filter sections, with and without corner-resonance:

Bode Plots

- A *Bode plot* draws the *magnitude frequency response* as *dB* versus some *log frequency*
- In the intro, we saw Bode plots showing the magnitude frequency-response of various Butterworth, Chebyshev, and Elliptic lowpass filters
- We can ask any good chatbot to make a Bode plot of `scipy.signal.freqs(B,A)`
- In *MATLAB* or *Octave*, we can say

```
sys = tf(1,[1,sqrt(2),1]);
bode(sys);
```

Bode Plots for Second-Order Butterworth Filters

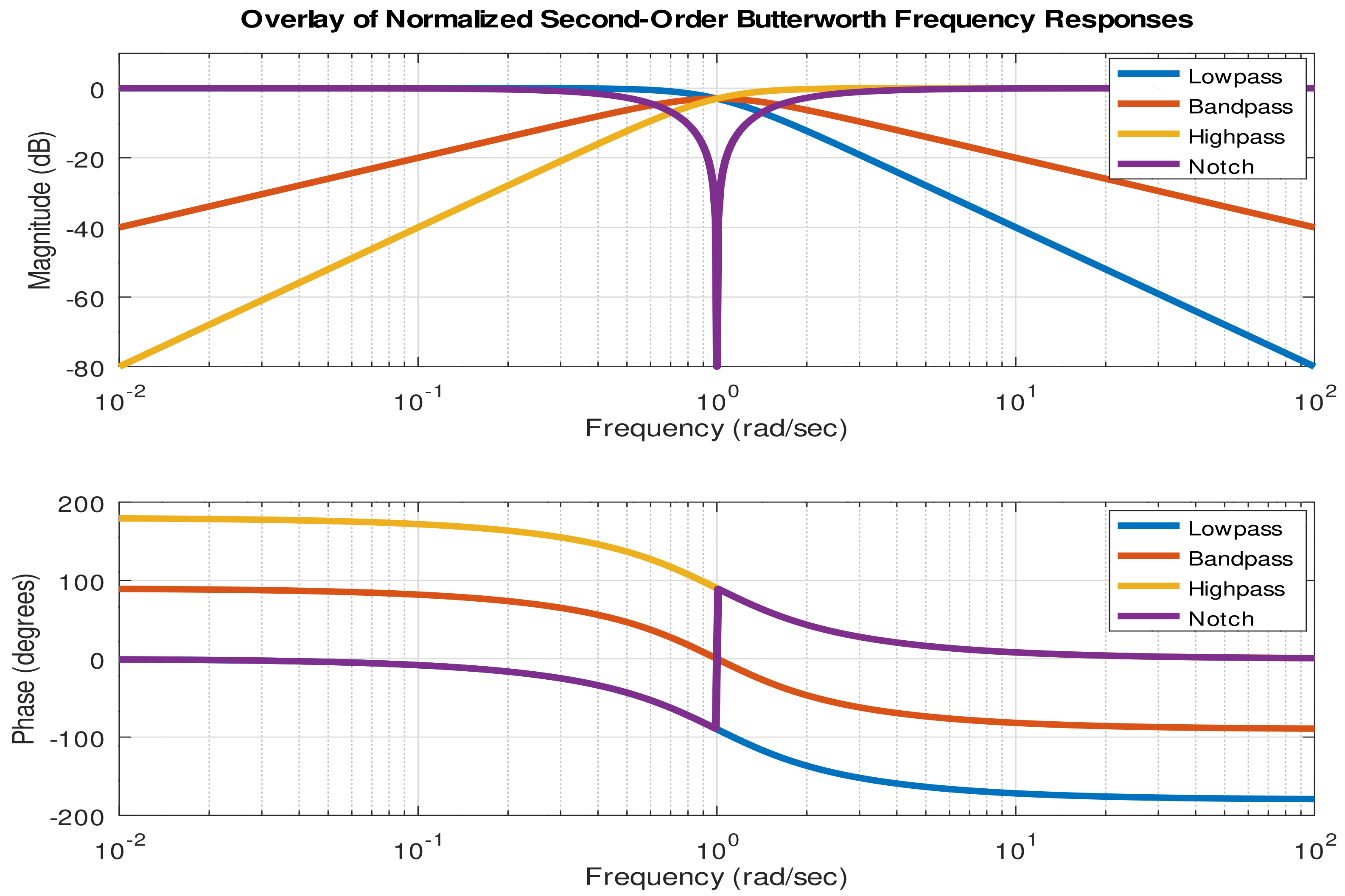
- Normalized Second-Order Butterworth Lowpass/Bandpass/Highpass/Notch:
  - `bode(tf([0 0 1],[1,sqrt(2),1])); % lowpass`
  - `bode(tf([0 1 0],[1,sqrt(2),1])); % bandpass`
  - `bode(tf([1 0 0],[1,sqrt(2),1])); % highpass`
  - `bode(tf([1 0 1],[1,sqrt(2),1])); % notch`

- These variations are normally brought out in second-order “state variable filters”

- Do not confuse “state variable filters” with “state variable representations” of linear systems: https://ccrma.stanford.edu/~jos/filters/State_Space_Realization.html https://ccrma.stanford.edu/~jos/StateSpace/State_Space_Models.html https://ccrma.stanford.edu/~jos/pasp/State_Space_Models.html

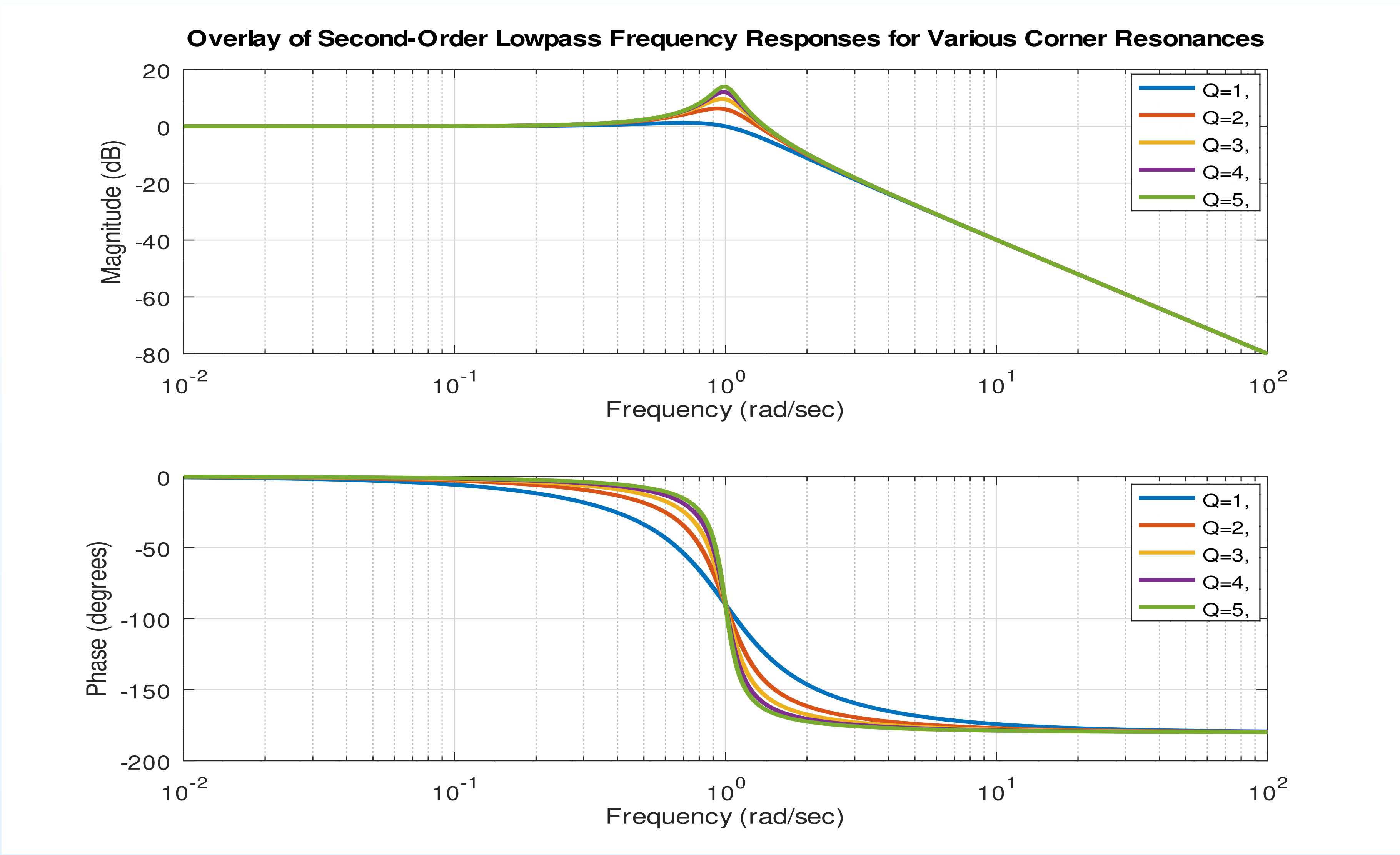
- The “state variable filter” is just a particular biquad structure: https://ccrma.stanford.edu/~jos/svf/

Bode Plots for Second-Order Butterworth Filters



Overlay of normalized 2nd-order Butterworth lowpass, bandpass, highpass, and notch.

Bode Plots for Second-Order Lowpass Filters with Corner Resonance



Overlay of second-order lowpass frequency responses for Q=1,2,3,4,5.

Chebyshev Filters

Chebyshev Filter Transfer Function

The Chebyshev filter power response is defined by:

$$|\mathcal{F}(j\omega)|^2 = \frac{1}{1+\varepsilon^2 C_N^2(\omega)}, \tag{9}$$

where $C_N(\omega)$ is an Nth-order, real-valued function of the real variable ω:

$$C_N(\omega) = \cos(N\cos^{-1}(\omega)). \tag{10}$$

Note that $C_0 = 1$ and $C_1 = \omega$.
We'll later use an intermediate complex variable $\phi = \cos^{-1}(\omega)$ to find the poles:

$$C_N(\omega) = \cos(N\phi), \quad \text{where } \omega = \cos(\phi) \tag{11}$$

We can check (`https://chatgpt.com/share/673fa853-e3f8-800f-a59c-d63738f6561e`) that

$$C_{N+1}(\omega) = 2\omega C_N(\omega) - C_{N-1}(\omega) \tag{12}$$

Thus, $C_N(\omega)$ is an Nth-order polynomial. We call it the Nth-order *Chebyshev Polynomial*.

Chebyshev Polynomial Examples

Using the recursion (12) we have

$$\begin{aligned} C_0 &= 1, \\ C_1 &= \omega, \\ C_2 &= 2\omega^2 - 1, \\ C_3 &= 4\omega^3 - 3\omega, \\ C_4 &= 8\omega^4 - 8\omega^2 + 1, \\ &\vdots \end{aligned} \tag{13}$$

Useful identities for developing these polynomials are

$$\begin{aligned} C_N^2 &= \frac{1}{2}[C_{2N} + 1], \\ C_{MN} &= C_M(C_N(\omega)) \quad \text{where M and N are coprime.} \end{aligned} \tag{14}$$

Poles of $\mathcal{F}(s)$

From (9) above, the poles of $\mathcal{F}(s)$ occur when

$$1+\varepsilon^2 C_N^2\left(\frac{s}{j}\right)=0.$$

Define $\cos(\phi)=s/j=-js$ and recall from (11) that $C_N(s/j)=\cos(N\phi)$:

$$0=1+\varepsilon^2 C_N^2\left(\cos(\phi)\right)=1+\varepsilon^2 C_N^2(N\phi).$$

Solving for ϕ yields N solutions ϕ_m:

$$\phi_m=\frac{1}{N}\arccos\left(\frac{\pm j}{\varepsilon}\right)+\frac{m\pi}{N},\quad m=0,1,2,\ldots,N-1.$$

The poles are then given by

$$s_m=j\cos(\phi_m),\ m=0,1,2,\ldots,N-1.$$

Chebyshev Poles in s Plane

The poles may be written more explicitly as[1]

$$\begin{aligned}
s_m &= j\cos(\phi_m)\\
&= \pm\sinh\left(\frac{1}{N}\operatorname{arcsinh}\left(\frac{1}{\varepsilon}\right)\right)\sin(\theta_m) + j\cosh\left(\frac{1}{N}\operatorname{arcsinh}\left(\frac{1}{\varepsilon}\right)\right)\cos(\theta_m)
\end{aligned}$$

with

$$\theta_m = \frac{\pi}{2}\frac{2m+1}{N}, \quad m = 0,1,2,\ldots,N-1.$$

Since an ellipse centered at $s=0$ in the complex plane can be described by

$$s = a\sin(\theta) + jb\cos(\theta), \quad \theta \in [-\pi,\pi]$$

(where a and b are the semi-axis lengths, one major and one minor when $a \neq b$), we find that the poles lie on an *ellipse* in s-space centered at $s=0$.

[1] See Wikipedia page for "Chebyshev Filter".

Chebyshev Poles in s and z Planes

Cursor Prompt (Claude 3.5 Sonnet):

> "This is great, now please add a function `plot_chebyshev_poles` that does the same thing for Chebeshev filter poles, which are on an ellipse inside the unit circle. If you need formulas, let me know."

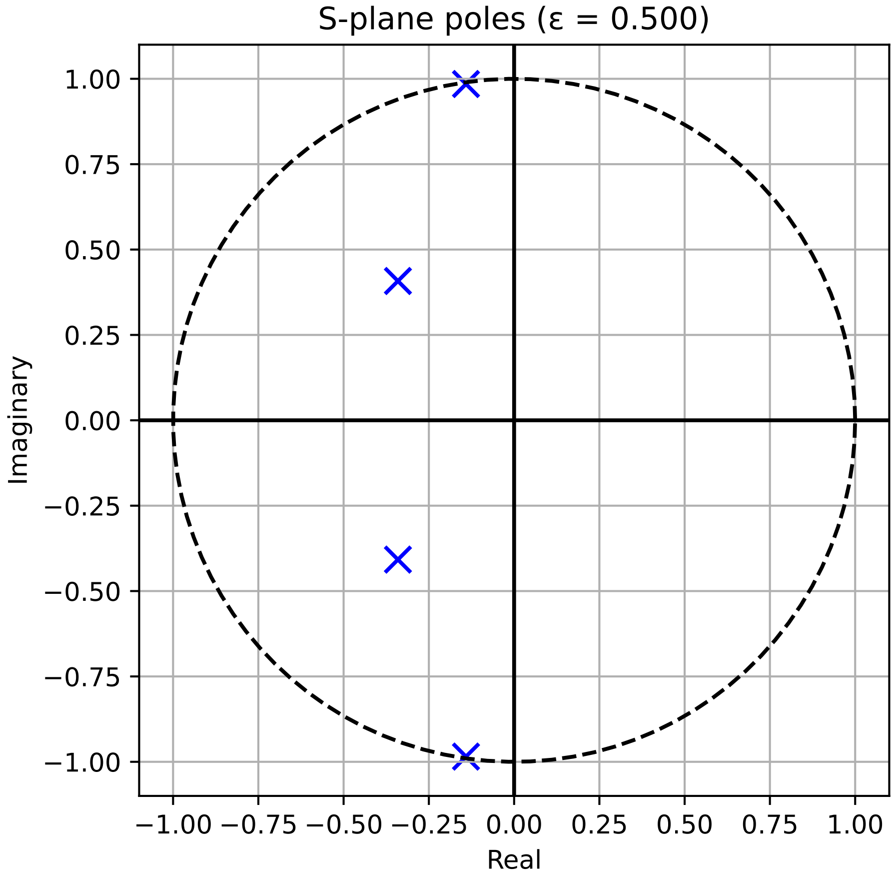
(It asked for formulas, which I pasted from this LaTeX source.)

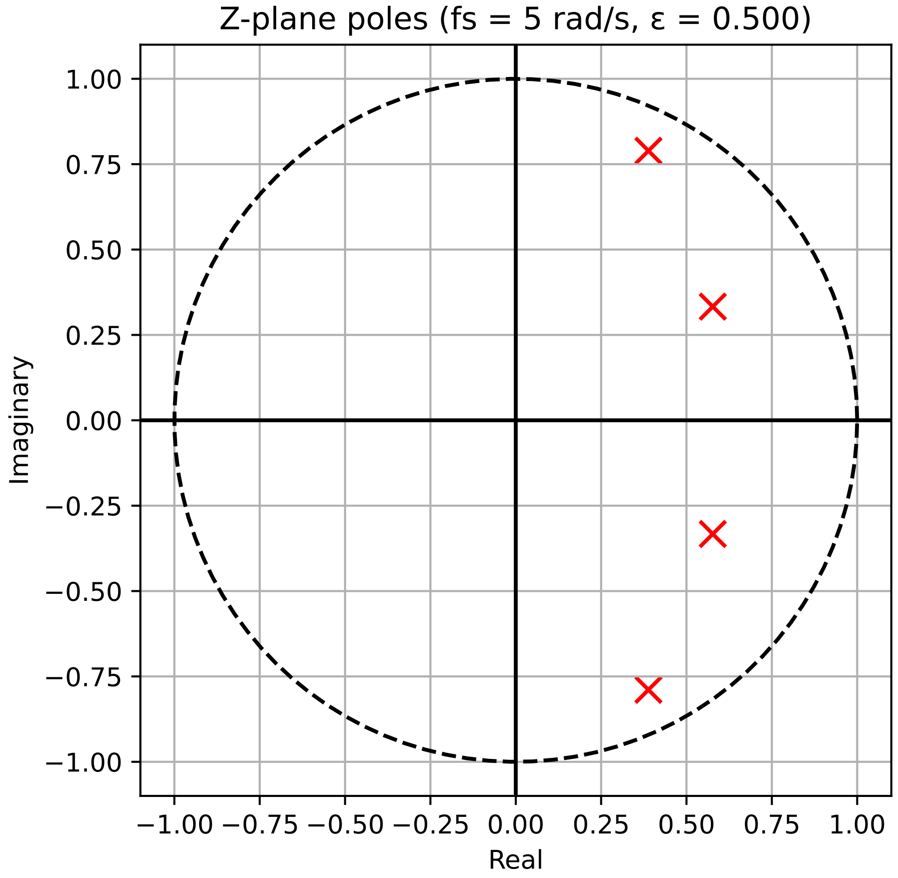
Needed Tweaks:

- Sign error in the pole real-parts (one-character fix)
- Keep the Butterworth test as an option instead of replacing it

Chebyshev Poles in s and z Planes Plotted

Result:





Elliptic Function (Cauer) Filters

Introduction to Elliptic Functions

Elliptic (Cauer) filters are based on *Jacobian elliptic functions*, which generalize the normal trigonometric and hyperbolic functions. The elliptic integral of the first kind is defined as

$$u(\phi, k) = \int_0^{\phi} \frac{dy}{\sqrt{1 - k^2 \sin^2(y)}} \tag{7.59}$$

The trigonometric sine of the inverse of this function is defined as the Jabocian elliptic sine of u with modulus k:

$$\text{sn}(u, k) = \sin(\phi(u, k)) \tag{7.60}$$

For details, see `https://en.wikipedia.org/wiki/Elliptic_filter`

Features of Elliptic Filters (Lowpass, Highpass, Bandpass, Bandstop):

- Chebyshev (equiripple) in both passband and stopband (*two* ripple parameters)
- Sharpest possible transition from passband to stopband or vice versa
- Much “phase distortion” (*e.g.*, “ringing”) at passband corners
- Optimal passband-ripple, stopband-ripple, and transition-width tradeoffs

General Digital Filter Design

Example Driving Problem: Real-Time Filter Design in an Audio Plugin

(Red-Bordered Buttons Added to **Plugin GUI Magic**'s Equalizer Example)

Methods for Arbitrary Filter Design

- Frequency Sampling
  1. *Draw* or *Load* Your Desired Magnitude Frequency Response
  2. Make it *Minimum Phase* (so the filter will be *causal*)
  3. Inverse-FFT gives the Desired Impulse Response (IR)
  4. "Window" the IR to the Affordable FIR length (smoothing the Frequency Response)
  5. Use *Convolution* to implement the FIR filter (typical for Amp Cabinets and such)

Methods for Arbitrary Filter Design, Continued

- Equation-Error Filter Design: Minimize $\|\hat{A}(\omega)H(\omega) - \hat{B}(\omega)\|$
  - E.g., `invfreqz` in MATLAB and Octave
  - We need C++ for an Audio Plugin!
    (or some easily embedded filter-design language)
  - AI Chatbots translate *well known languages* to C++ very well
  - They also write good starting *unit tests*
  - Speed Bumps:
    - MATLAB is proprietary (and no longer even precisely documented)
    - Octave is GPL (but contributing authors could be asked for permission)
    - Python is mostly BSD, but has no `invfreqz` yet in `scipy.signal`
  - **Plan:** Implement `invfreqz` from scratch in Python and translate to C++
  - **Method:** Paste the algorithm description[2] into Claude 3.5 Sonnet and debug
  - **This actually worked!**

---

[2] https://ccrma.stanford.edu/~jos/filters/FFT_Based_Equation_Error_Method.html

Claude 3.5 Sonnet Converts LaTeX Description of `invfreqz` to Python

1. *Prompt 1:*

   Following is a LaTeX description of a fast equation-error algorithm. Please write a Python implementation.
   <LaTeX source of algorithm description>

2. *Prompt 2:*

   Write a separate test program in Python which uses `` `scipy.freqz` `` to generate three different test examples of progressing complexity. That way, the original and estimated filter coefficients can be compared. A good source of example starting filters would be `` `scipy.signal.butter` `` and `` `scipy.signal.cheby1` `` etc.

3. This was the starting test program for the one in my `scipy` fork:
   `https://github.com/josmithiii/scipy/blob/jos/scipy/signal/test_invfreqz_jos.py`

Claude 3.5 Sonnet Converts a *Paper Abstract* to Working Python

Prompt: Write a Python function that designs a *spectral tilt filter* as described in this paper abstract:[3]

> We derive closed-form expressions for the poles and zeros of approximate fractional integrator/differentiator filters, which correspond to spectral roll-off filters having any desired log-log slope to a controllable degree of accuracy over any bandwidth. The filters can be described as a **uniform exponential distribution of poles along the negative-real axis of the s plane, with zeros interleaving them. Arbitrary spectral slopes are obtained by sliding the array of zeros relative to the array of poles, where each array maintains periodic spacing on a log scale.** The nature of the slope approximation is close to Chebyshev optimal in the interior of the pole-zero array, approaching conjectured Chebyshev optimality over all frequencies in the limit as the order approaches infinity. Practical designs can arbitrarily approach the equal-ripple approximation by enlarging the pole-zero array band beyond the desired frequency band. The spectral roll-off slope can be robustly modulated in real time by varying only the zeros controlled by one slope parameter. Software implementations are provided in matlab and Faust.

[3] `https://ccrma.stanford.edu/~jos/spectilt/spectilt.pdf`

Why Python?

- The test `__main__` block can conveniently use `numpy`, `scipy`, and `matplotlib` functions for test displays and subsequent interactive development
- Chatbots:
  - are trained on a *lot* of Python, and it's a relatively simple language,
  - are *not yet good* at signal processing (even simple polynomial algebra), and
  - tend to fall apart on low-level signal-processing details
- I influence them to work in terms of *well documented high-level APIs* such as functions in `scipy.signal` rather than writing C++ from scratch
- Translation from Python to C++ has been mostly smooth
- `Eigen3` gets used a lot

`invfreqz.py` Today

`invfreqz.py` is working now in the jos `scipy` fork at
`https://github.com/josmithiii/scipy/blob/jos/scipy/signal/test_invfreqz_jos.py`
(pull-request in preparation)

Features:

- New `min_phase` option for creating minimum phase desired frequency response
- New `stabilize` option for reflecting unstable poles into the unit circle
- New `method` argument for selecting other methods besides equation-error:
  - Equation-error method (default)
  - Steiglitz-McBride (original iterative method)
  - Prony's method (least-squares numerator)
  - Padé-Prony method (impulse-response-matching numerator)
  - Maybe: "Recursive Gauss-Newton iterations" [Hessian(n) $\approx \sum_n \nabla_n \nabla_n^T$)]
  - Maybe: *Neural map* from desired frequency response to starting poles and zeros
- All but Steiglitz-McBride are passing their unit tests
- It remains to decide what to finally do and integrate the proposed final version into `_filter_design.py` for a `scipy` pull request

`scipy.cpp` Today

Since Claude uses `scipy.signal` functions in its generated Python, we need those translated to C++ as well. Translated so far by Claude (most were fast):

- `tf2zpk` - convert transfer function to zero-pole-gain (ZPK) representation
- `zpk2tf` - inverse of tf2zpk
- `tf2sos` - convert transfer function to second-order-sections (sos)
- `sos2tf` - inverse of tf2sos
- `zpk2sos` - zero-pole-gain (ZPK) directly to SOS
- `roots` - compute the roots of a polynomial (uses Eigen3)
- `bilinear` - convert analog IIR filter to digital using bilinear transform
- `bilinear_zpk` - bilinear transform for zeros, poles, and gain
- `lp2lp_zpk` - lowpass to lowpass frequency scaling for analog zeros, poles, and gain
- Unit Tests for all (`Catch2`) — *This is very important* — *Claude can write most of them*
- Status:
  - Working through what's needed now in `_filter_design.py` and its dependencies
  - A complete `scipy.signal.cpp` would nice to complete from there
  - Other `scipy` subirectories, such as `fft` and `linalg`, are in much better shape

General Filter Design Summary

- Translating Python to C++ for real-time use is greatly facilitated by Chatbots
- Claude 3.5 Sonnet has been the clear winner for me
- They all struggle with sample-level signal processing, and polynomial algebra
- Several `scipy.signal._filter_design` functions are done and tested
- In general, Python is a good intermediate language for new C++ DSP functions
  - Pushes chatbots away from sample-level code
  - Facilitates visual test plots using `matplotlib` etc.
  - Encourages simpler C++ using Eigen3 etc.
- `invfreqz` is now available in Python on GitHub
- `scipy.signal.cpp` seems about half done
- These overheads (including all *links*) are available on the JOS Home Page (as well as the ADC website)

Summary of Resources Online

- JOS Home Page (Videos, Overheads, including these):
  `https://ccrma.stanford.edu/~jos/`

- Equation-Error Minimization for Filter Design:
  `https://ccrma.stanford.edu/~jos/filters/FFT_Based_Equation_Error_Method.html`

- `invfreqz` for Python in JOS *scipy* fork:
  `https://github.com/josmithiii/scipy/blob/jos/scipy/signal/test_invfreqz_jos.py`

- Spectral Tilt Filters:
  `https://ccrma.stanford.edu/~jos/spectilt/spectilt.pdf`