ECS7012P Music and Audio Programming • Spring 2023

Assignment 1: Synth Filter

Handed out: Monday 6 February 2023
Report and Code Due: 10:00, Friday, 3 March 2023

Introduction:

Your task is to implement a resonant lowpass filter which models a famous design from the early days of analog synthesis.

Analog synthesisers created by Robert Moog, Don Buchla and other pioneering inventors flourished starting in the 1960's and 1970's. Many of these synths used a process called *subtractive synthesis* where a spectrally rich oscillator (i.e. one containing many harmonics) passes through a *voltage-controlled filter* (*VCF*) which amplifies certain parts of the spectrum while attenuating others. Changing the parameters of the filter changes the *timbre* (tone quality) of the resulting sound, giving rise to many of the iconic sounds of this era.

Today, most synths use digital signal processing rather than analog components, but digital emulation of these early analog designs remains popular. Many research papers have been written about how to emulate the particular idiosyncrasies of famous filter designs by Moog and other manufacturers.

In this assignment, you will create a digital implementation of a common type of filter used in these applications, the *resonant lowpass filter*. Like all lowpass filters, it should pass through low frequencies while attenuating or blocking high frequencies. The term *resonant* refers to its high Q-factor, which produces an audible peak in the spectrum around the point of the filter cutoff, which for decades musicians have found appealing.

Specifically, you will be implementing an emulation of the classic Moog VCF based on a simplified digital model by Vesa Välimäki and Antti Huovilainen [1]. This handout will take you through the mathematical and programming steps you need to calculate the filter outputs, implement the equations in C++ code on the Bela platform, and measure the filter's performance.

Note: I strongly recommend that you complete Lecture 3b ("Filters"; Lecture 8 on YouTube) prior to completing this assignment: **https://www.youtube.com/watch?v=XVOdqJy-Rfg**

[1] Välimäki, V. and Huovilainen, A., 2006. Oscillator and filter algorithms for virtual analog synthesis. *Computer Music Journal*, *30*(2), pp. 19-31. **Please read p. 26 onward.** https://www.jstor.org/stable/pdf/3682001.pdf

Required Materials:
• Bela Kit and cables
• Code template from QMPlus (contains basic audio I/O and oscillator but no filters)

Instructions:

1. Overview of the task

In this assignment you will implement a digital emulation of a Moog voltage-controlled filter (VCF). The Moog filter is a fourth-order resonant lowpass filter with adjustable *cutoff frequency* and *resonance* (Q). In the original analog circuit, and in most digital emulations, the Moog filter is implemented as four first-order filters in series, with global feedback around the whole structure to produce the resonance:

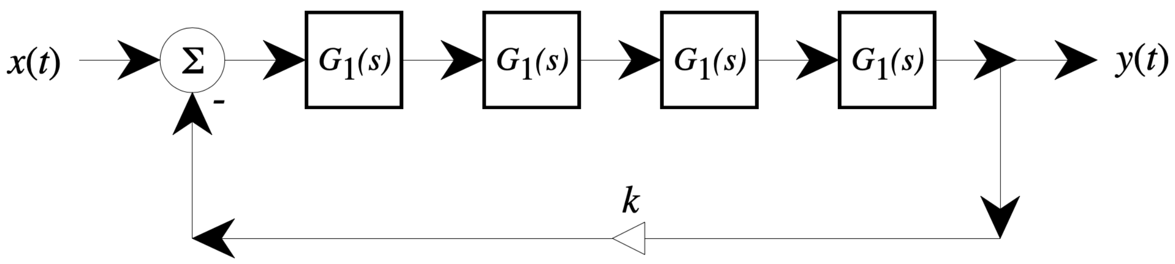


Figure 1. Block diagram of the original (analog) Moog VCF. From T. Stilson and J. O. Smith, "Analyzing the Moog VCF with Considerations for Digital Implementation". Proceedings of the International Computer Music Conference, 1996. https://ccrma.stanford.edu/~stilti/papers/moogvcf.pdf

A distinctive characteristic of the Moog VCF is its *nonlinearity*. Unlike familiar linear time-invariant (LTI) signal processing systems, the Moog filter introduces distortion in its output. This distortion can have the effect of introducing *harmonics* (new frequencies that are multiples of existing frequencies in the input signal). Although this behaviour is not "accurate" filtering in a scientific sense, it has a distinctive musical character that remains highly sought-after and frequently emulated. However, emulating the nonlinearity digitally can be very difficult!

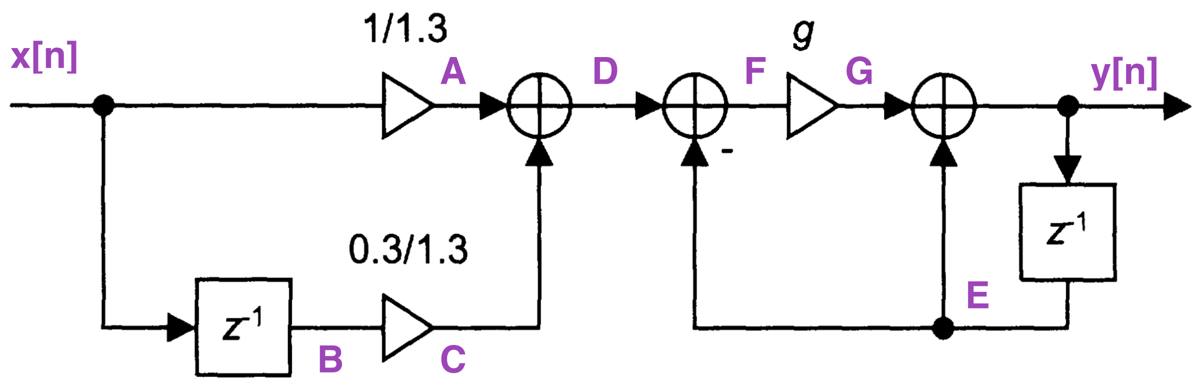
Several digital models of the Moog VCF have been proposed in the research literature. This assignment will focus on a model from a 2006 paper by Vesa Välimäki and Antti Huovilainen (see Introduction above). The next steps will guide you through the implementation.

2. Oscillator

To test your filter you will first need a signal source to use as its input. The project code provides a choice of two oscillators as input: a sine wave, which is useful for measuring frequency response and distortion, and a sawtooth wave, which has rich harmonic content that makes the effect of the filter clearly audible.

You can choose between the oscillators inside of `render()`; it is configured for a sine by default. The frequency of the chosen oscillator can be adjusted using a slider in the web browser. Upload the Assignment 1 project to your Bela board, run it and verify that you can adjust the oscillator frequency by loading the GUI from the Bela IDE (click the icon on the bottom that shows a set of sliders).

3. Implementing one first-order section



Välimäki and Huovilainen use the following model (derived from Stilson and Smith 1996, cited above) for each first-order section of their filter:

Figure 2. One first-order section of the Moog VCF emulation. From Välimäki and Huovilainen (2006).

3a. Calculating the formula for the first-order section

Above, in purple, I have annotated the block diagram with letters representing each node. In your report, write equations for the value of the signal at each labelled node. Remember that the triangular block corresponds to a gain (i.e. a multiplication) and the block labelled z^{-1} represents a unit delay. (Don't forget to check for minus signs on the summing blocks!) Based on those equations, write the complete formula for y[n] based on y[n-1], x[n] and x[n-1]. Recall that the general formula for a first-order IIR filter is:

$$y[n] = b_0 x[n] + b_1 x[n-1] - a_1 y[n-1]$$

Thus, once you have the equation, you can calculate the coefficients a_1, b_0, b_1 to use in your code.

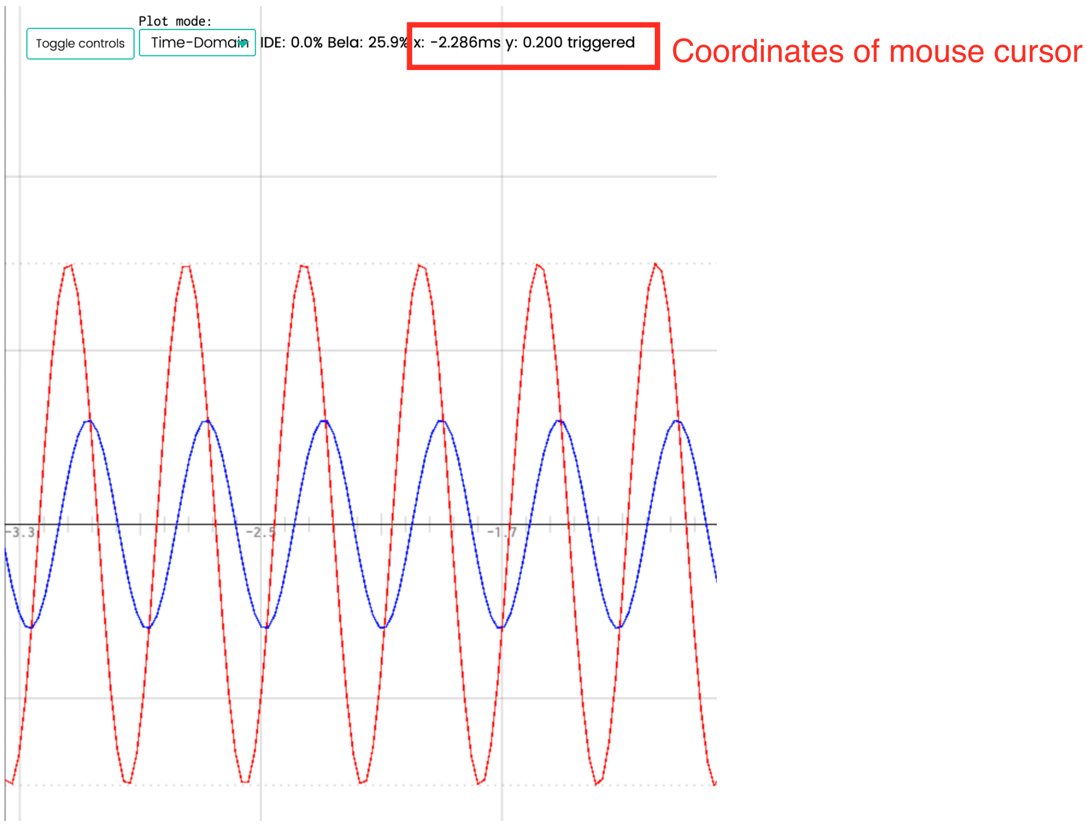
3b. Implementing the filter in code

In your Bela project, implement a single first-order filter, which takes as input the signal from the oscillator, and sends its output to the audio outputs and the Bela oscilloscope. Your code should calculate the coefficients inside the function `calculate_coefficients()`, and then should apply the filter inside of `render()`. *Hint: Don't forget to declare any global variables you need to keep track of previous values of x[n] and y[n], following the Filters lecture.* Give your variables sensible names reflecting how they appear in the equation, so your code is easier to read. Calculate *g* based on the second argument (`frequencyHz`) to `calculate_coefficients()`, using the formula: $g = 2\pi f_c / f_s$

3c. Measuring filter performance

Use the sine wave oscillator as the input to your filter. Inside of `render()`, set the latter two arguments to `calculate_coefficients()` to 1000 and 0 (the third argument isn't used for now; we will make the parameters adjustable later). Run the project, open the Bela GUI to see the slider for adjusting oscillator frequency (leave the oscillator amplitude at its default value of 0.5), and open the oscilloscope so you see the input and output waveforms. Take measurements of the filter gain at a series of frequencies below and above 1000Hz.

Hint: at the top of the oscilloscope screen, you can see the X and Y values for the mouse cursor, which you can use to find the peak value of the output sine wave (blue plot).



Remember that you want the ***gain*** *of the filter, not the raw amplitude of your output, so don't forget to normalise by the amplitude of your input oscillator (whose amplitude you can read from the GUI or from the red plot).*

Figure 3. Reading values from the Bela oscilloscope. Hold the mouse over the peak of the signal of interest and note the value at the top of the screen.

Make a plot of gain versus frequency for this filter (ideally plotted on a logarithmic Y axis) and include it in your report. At what frequency (approximately) do you find the -3dB point?

Next, change the second argument to `calculate_coefficients()` to be 4000. Now measure the -3dB point again. What value do you find? (You don't need to make another frequency response plot.)

3d. Improving the frequency calculation

Välimäki and Huovilainen observe that the value of g in Figure 2 (above) relates nonlinearly to the measured cutoff frequency (-3dB point) of the filter. The simple calculation $g = 2\pi f_c/f_s$ only produces an accurate approximation of the desired cutoff frequency for low values of f_c. They propose a polynomial model which improves the fit:

$$g = 0.9892\omega_c - 0.4342\omega_c^2 + 0.1381\omega_c^3 - 0.0202\omega_c^4 \quad \text{where} \quad \omega_c = 2\pi f_c / f_s$$

Implement this equation inside of `calculate_coefficients()`. *(Hint: you can use the* `powf()` *function to implement X to the power of Y.)*

Leaving the filter frequency at 4000, run the project again and measure the -3dB point again. Is it more accurate?

Notice that so far, we have not yet seen a way to adjust the *resonance* (Q factor). This comes later, following the introduction of feedback in step 6.

4. Fourth-order filter without feedback

Next, create a fourth-order filter by putting four of the first-order blocks from the previous step in series. Each first-order section has the same coefficients, but remember that you will need separate variables to keep track of the previous inputs and outputs for each section. *(Hint: you might find it helpful to use arrays for this).* Optionally, you could consider making a simple C++ class to encapsulate the first-order filter, including its coefficient calculations.

Once you have completed this step, run the project again with the same settings as step 3d (4000Hz filter frequency). Make another frequency response plot using the same procedure as steps 3c-3d (with the filter set to 4000Hz).

In step 3d, you identified the frequency where the gain was down by 3dB. Take a measurement of this new fourth-order filter at the same frequency. Now what is the gain at this frequency? Why?

5. Add nonlinearity

As discussed above, the nonlinearity of the Moog VCF is a crucial part of its sound. In this model, the nonlinearity of the analog circuit is approximated by a hyperbolic tangent (*tanh*) function. On Bela the most efficient implementation of this function is called `tanhf_neon()`, optimised for the NEON floating-point unit of the Bela board which we will discuss toward the end of this module. Add this function before the input to the filter (i.e. the oscillator goes into `tanhf_neon()`, and the result goes into the filter). *(Warning: in case you are now tempted to try* `powf_neon()` *for the calculation in step 3d -- don't! This function isn't accurate for inputs less than 1.0.)*

Run the project with a sine wave input and set the oscillator frequency to 500Hz (filter frequency still 4000Hz). Gradually increase the oscillator amplitude to its maximum value in the GUI and watch the output in the Bela oscilloscope. What happens to the shape of the blue signal as you increase the amplitude?

Next, in the oscilloscope, in the upper-left corner of the window, click the box labelled "*Time Domain*" and change it to "*FFT*". This shows a real-time spectrum (frequency domain plot) of your signal. Click the *Toggle Controls* button and change the *Y-Axis* setting to *Decibels*. What do you see when the sine wave has a low amplitude? What happens as you increase the amplitude? Why?

6. Add feedback

In this step we will implement the complete model by Välimäki and Huovilainen:

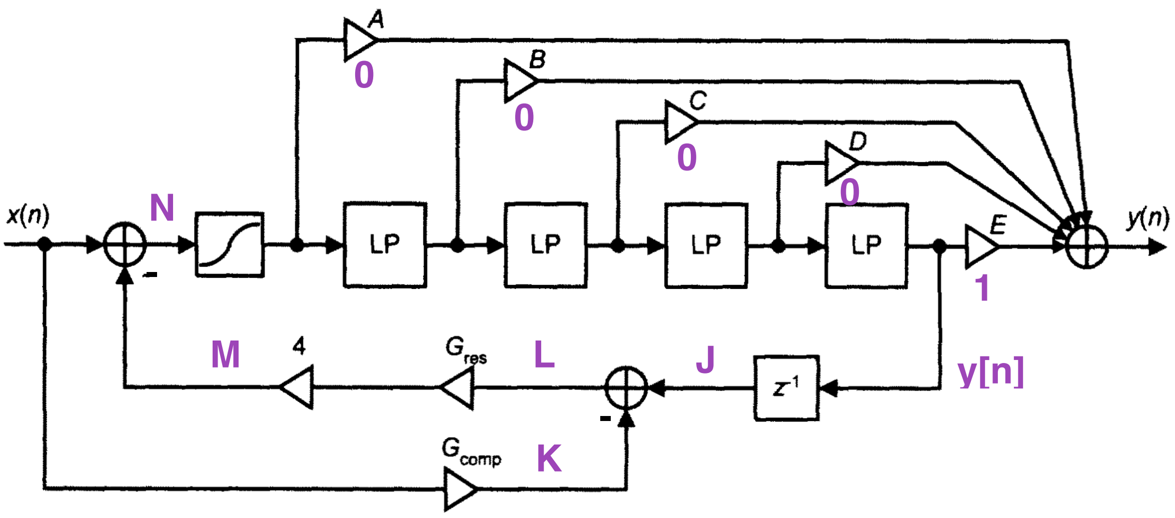


Figure 4. Digital model of the Moog VCF, including four first-order sections, a nonlinearity and feedback. From Välimäki and Huovilainen (2006).

In Figure 4, notice the feedback loop from the output of the fourth-order filter back to the input, which includes a delay block (z^{-1}). This delay is not present in the original analog circuit, but without it, the filter becomes difficult or impossible to implement in the digital domain because it would contain what is called a *delay-free loop*, where we need to know the output of the filter before we can calculate its input (and vice-versa)! Such systems usually have no closed-form mathematical solution and can only be solved with computationally expensive iterative approximation. The z^{-1} block introduces some inaccuracy in the emulation but it makes our filter much easier to calculate.

6a. Calculating the formula for the feedback loop

Referring to Figure 4, in the Moog VCF, only the final low-pass filter section goes to the output. In order words, $E = 1$, $A = B = C = D = 0$. (See the bonus step of this assignment for an exploration of other configurations.) On that basis, we know that $y[n]$ is the input to the feedback loop at the bottom of the diagram.

Referring to the purple letters in the diagram, write out the formula for each letter in terms of x, y, G_{res} and G_{comp}. In particular, pay attention to the value at node N as this is what will go into the nonlinearity in your filter. *(Hint: don't forget to check the minus signs on the summing junctions!)*

6b. Implementing the feedback loop in code

In your Bela project, implement the feedback loop from Figure 4 in your `render()` function. Compared to step 5, you should not need to change any of the first-order filter coefficients nor change the nonlinearity. Rather, the input to the nonlinearity should be the formula you derived in step 6a.

Declare a new global variable to hold the value of G_{res}. The value of G_{comp} is fixed by design at 0.5. Initially, try your filter with 1000Hz cutoff frequency and $G_{res} = 0.75$. (In the next step we will make them adjustable.)

6c. Evaluating filter performance

Use a sine wave oscillator with amplitude 0.1 (this will reduce the effect of the nonlinearity, making the resonant peak more pronounced). Take a frequency response plot similar to step 3c. (You might need to change the Bela oscilloscope from *FFT* back to *Time-domain*.) What frequency has the peak magnitude response? What is the gain (in decibels) at the peak?

7. Make filter adjustable

In this step, we will add GUI controls to adjust the cutoff frequency and resonance of the filter. Optionally, instead of using the GUI, you could use physical potentiometers on your breadboard, connected to the Bela analog inputs, to adjust the filter parameters.

7a. Adjusting cutoff frequency

In step 3 we passed a cutoff frequency to `calculate_coefficients()`. Now we will make that number adjustable in the GUI. In `setup()`, after the lines declaring the controls for the oscillator, add another line to make a new control for filter cutoff frequency. Set its range to 100-5000Hz, with a default value of 1000Hz. At the top of `render()`, query the value of the new control analogously to the other controls. Pass the cutoff frequency into `calculate_coefficients()`. Leave G_{res} at 0.75 for now.

To test whether this adjustment works, run the project with a sawtooth oscillator input. (Try a low oscillator frequency, say 100Hz, and an amplitude of 0.3, for a clear result.) How does the sound change when you sweep the filter frequency up and down?

7b. Adjusting resonance

Using the same procedure as 7a, add one more GUI control to adjust the filter resonance. The slider should take a value between 0 (no resonance) and 1 (maximum resonance). However, in `render()`, don't use this value in your equations just yet!

Välimäki and Huovilainen found that the raw value (which they call C_{res}) maps imperfectly onto the actual measured resonance of the digital filter. They calculate a cubic polynomial to correct for the error they observe:

$$G_{res} = C_{res}(1.0029 + 0.0526\omega_c - 0.0926\omega_c^2 + 0.0218\omega_c^3)$$

In the above equation, C_{res} is the value you get from the GUI control and G_{res} is the value you should pass to the equation you implemented in step 6b. ω_c is the angular frequency between 0 and 2π, which you first calculated in step 3d (**not** the raw value in Hertz!).

Implement the above calculation inside `calculate_coefficients()`, passing in the value of C_{res} from the slider (this goes in the third argument to the function, which has been unused until now). The value of G_{res} should be stored in the global variable you declared in step 6b.

Once you have finished implementing this, run the project again with a sawtooth wave input and the oscilloscope set to *FFT* mode. Set the oscillator to around 100Hz, amplitude 0.3, and the filter to 2000Hz and resonance 0.9. Include a screenshot of the plot in your report. *(Hint: in the oscilloscope controls there is a button "Pause plotting" that you can use to freeze the display for taking a screenshot.)* Can you see the resonant peak?

7c. Pushing the limits

Now try changing the range of the resonance slider to go from 0 all the way up to 1.1. Run the project again and push the resonance all the way up to the maximum. Try adjusting the filter cutoff frequency, and then turn the oscillator amplitude down to 0. Something unusual happens. What is it, and why do you think it happens? *(Hint: this was a feature of the original analog Moog VCF too!)*

8. Optional bonus step (up to 5% extra marks): implement other filter types

In the Välimäki and Huovilainen paper, Table 1 (p. 29) shows different choices of the gain coefficients A-E from Figure 16 of the paper (which is reproduced as Figure 4 of this assignment). By setting these coefficients to different values, you can achieve different types of filter, including lowpass, bandpass and highpass, either 2nd-order or 4th-order.

As a bonus, make a simple interface (e.g. another slider or a physical button on the breadboard) which lets you choose between these different options. Show a spectrum plot (in the same style as step 7b) of one other setting to show that this works.

For your own exploration (not marked):

You have now finished the emulation of the Moog VCF! You might find it fun to try it out in an actual synth context. Lectures 15-16 in the YouTube series, which we will cover around Week 8 of this module, introduce the MIDI protocol and build up a simple synth. When we get there (or if you feel like watching ahead), try your new Moog emulation in place of the simpler resonant filter that is used in that lecture.

The quality of the effect will also be improved if you use a band-limited sawtooth oscillator in place of the simple one we use in this assignment. The very same paper that this filter is taken from includes techniques for band-limited oscillators. Alternatively, see the QMplus materials for Week 3 for some extra suggested readings around band-limited synthesis.

Your submission should consist of the following:

1. Your **source code**. All `.cpp` and `.h` files that you edit or create for the project including `render.cpp`.
   - Don't forget to comment your code! Uncommented or illegible code will receive a reduced mark.
   - Please submit a zip file containing your source code and your report via QMPlus.
2. A **3 to 4-page report (PDF format)**. Include the following information (though your report does not have to be structured in this way):
   - Equations for different nodes of the block diagrams (for certain steps)
   - Plots of frequency response or spectrum, as requested in certain steps
   - Answers to other questions asked in certain steps
   - Cite your sources! For all assignments, any code or designs found to come from another source without attribution will be treated as a case of plagiarism, and may be referred to the university for further action.

Marking Criteria:

Theory	Demonstrates an understanding of the theory being implemented.	30%
Implementation	Code functions correctly, is well-commented, and design decisions are explained/justified in the report.	30%
Evaluation	Evidence demonstrating the implementation's success and discussion of any weaknesses.	30%
Report	Professional presentation of a technical report; sources are cited; figures are properly referenced and of sufficient resolution to be legible.	10%

Useful References:

Papers:

[1] Välimäki, V. and Huovilainen, A., 2006. Oscillator and filter algorithms for virtual analog synthesis. Computer Music Journal, 30(2), pp. 19-31. https://www.jstor.org/stable/pdf/3682001.pdf

[2] T. Stilson and J. O. Smith, "Analyzing the Moog VCF with Considerations for Digital Implementation". Proceedings of the International Computer Music Conference, 1996. https://ccrma.stanford.edu/~stilti/papers/moogvcf.pdf

[3] Huovilainen, A., 2004. Non-linear digital implementation of the Moog ladder filter. In Proceedings of the International Conference on Digital Audio Effects (DAFx), pp. 61-64. http://ucep.ece.gatech.edu/wp-content/uploads/sites/466/2016/11/P_061.pdf

Analog Filter Design: Hank Zumbahlen, *Linear Circuit Design Handbook*, ch. 8. Online: http://www.analog.com/library/analogdialogue/archives/43-09/edch%208%20filter.pdf

Bilinear Transformation: Julius O. Smith, *Physical Audio Signal Processing*. Online: https://ccrma.stanford.edu/~jos/pasp/Bilinear_Transformation.html

Digital Filter Design/Implementation: Julius Smith, *Introduction to Digital Filters*. Online: https://ccrma.stanford.edu/~jos/filters/

Modular Synthesis: Chris Meyer, *Learning Modular*. (This is written for beginning modular synth players rather than people who want to create synths using DSP, but it presents the concepts.) Online:
https://learningmodular.com/
About VCFs: https://learningmodular.com/glossary/vcf/

General DSP reference: A. Oppenheim & R. Schafer, *Discrete-Time Signal Processing.* Third edition. Pearson: 2011 (look in library).