MUS424L:
Signal Processing Techniques for Digital Audio Effects `Handout #1`
Jonathan Abel, Mark Rau `January, 30 2024`

Homework #1: Time-Varying Resonant Filters [25 points]
Due Date: January 18, 2024

Time-Varying Resonant Filters

Submission Instructions

Submit via Canvas. Create **a single compressed file** (`.zip`, `.tar` or `.tar.gz`) containing all your submitted files. Upload the compressed file to the homework submission dropbox. Name the file using the following convention:

```
<suid>_hw<number>.zip
```

where `<suid>` is your Stanford username and `<number>` is the homework number. For example, for Homework #1 my own submission would be named `dpberner_hw1.zip`.

For coding problems (either C++ or Matlab code), submit all the files necessary to compile/run your code, including instructions on how to do it. In case of theory problems, submit the solutions in **PDF format only**. LaTeX or other equation editors are preferred, but scans are also accepted. In case of scanned handwriting, make sure the scan is legible.

Lab1: Time-Varying Resonant Filters

In this lab, we will explore resonant and time-varying filters. The provided file contains a plug-in incorporating a filter whose response can be controlled by a slider. The code supplied in the handout contains a design function for a resonant lowpass filter, with the cutoff frequency for the filter being determined by the position of the control slider.

The assignment will involve replacing the included design function with a new design which models the responses measured from a guitar "wah" pedal. The measured responses can be approximated using a peaking filter.

Modeling the measured functions can be carried out using Matlab, and a Matlab script is included in the supplied materials to plot the measured and modeled responses. The modeling will be carried out in Problem 1 of the assignment. Mark will work on converting this to python, but if anyone gets there before him, please share with others!

Once the modeling has been completed, the plugin must be edited to include a *warped* bilinear transform function, in order to create discrete-time filters which approximate the modeled responses at the sampling rate for the plug-in. This will be accomplished in Problem 2.

In Problem 3, the plug-in must be edited so that as the control position is changed, the designed filter coefficients are varied smoothly towards the desired values. Two methods of coefficient smoothing will be explored in the problem.

For all questions, please turn in your code, any audio files rendered for the problem, and a list of parameters chosen to produce the desired output.

Problem 1. [25 Points]

The resonant peaking filter is specified by its center frequency ω_0, resonance Q, and gain γ. Its transfer function is given by

$$H(s) = \frac{\gamma \cdot s/\omega_0}{(s/\omega_0)^2 + \frac{1}{Q}(s/\omega_0) + 1}. \tag{1}$$

The MATLAB script `modelwahfilters.m` will load measured data from the file `wahtransferfunctions.mat`. The data file includes measured responses for eleven positions of the "wah" pedal. The script plots the magnitude response for the measurement specified by the variable `measurementnumber`, which should be set within the range $1-11$ to see the particular measurement being modeled. The measured data will be plotted in blue.

The response for a peaking filter will be overlaid in red, with the modeled filter specified by the MATLAB variables `f0`, `Q`, and `gamma`.

1(a). [**10 Points**] Choose parameter values `f0`, `Q`, and `gamma` to match the eleven measured transfer functions as closely as possible. Note that because of the way the mechanical action of the pedal works, the first several measured responses are very similar to each other.

Turn in a plot showing the measured and modeled responses, overlaid.

1(b). [**15 Points**] Populate the function `DesignWahFilter` in the source file `PluginProcessor.cpp` within the plug-in code. **Note:** *All places within the source code that must be edited are tagged with the string* `'Lab1_Task'` *within the comment field. Searching within the project for this string will reveal all functions which must be edited to carry out the assignment.*

The function takes as input the `double` argument `'position'`. This argument ranges in value from 0.0 to 1.0 and represents the position of the control slider. This range is identical to the range of the variable `pedal` used in the MATLAB script to identify the pedal positions corresponding to the measurements which were modeled in Problem 1(a).

When writing the code for your function, the design can be linearly interpolated between the eleven measurements, with the measurements themselves falling on `position` values $0.0, 0.1, 0.2, ...$

For example, if your MATLAB fit corresponding to `pedal` value 0.4 resulted in values ω_4, Q_4, and γ_4, then your `C++` function should use those values when called with `position`$= 0.4$.

When called with `position`$= 0.44$, the values used would be:

$$\begin{aligned}\omega &= 0.4 * \omega_5 + 0.6 * \omega_4\\ Q &= 0.4 * Q_5 + 0.6 * Q_4\\ \gamma &= 0.4 * \gamma_5 + 0.6 * \gamma_4\end{aligned}$$

The computed coefficients for the peaking filter should be assigned to the `b[]` and `a[]` members of the instance `wahFilter` of the struct `DirectBiquad`, *i.e.*

$$\begin{aligned}\texttt{wahFilter.b}[0] &= b2\\ \texttt{wahFilter.b}[1] &= b1\\ \texttt{wahFilter.b}[2] &= b0\\ \texttt{wahFilter.a}[0] &= a2\\ \texttt{wahFilter.a}[1] &= a1\\ \texttt{wahFilter.a}[2] &= a0\end{aligned}$$

Note that when this function is called, it in turn calls the `DirectBiquad` member function `Bilinear`, which calculates the discrete-time coefficients for the filter. Make sure to leave this code in place. For an example of how this should look, refer to the function `DesignResonantLPF` directly above the function you are writing.

Make sure to uncomment the call to `DesignWahFilter` from the function `sliderValueChanged` in the file `Controls.cpp`. Also, comment out the call to `DesignResonantLPF`, since it will be replaced with your function. This section of code can be found by searching the project for the string `Lab1_Task`.

Problem 2. [25 Points]

The *bilinear transform* is defined by

$$s \to \left(\frac{2}{T_d}\right)\frac{1 - z^{-1}}{1 + z^{-1}}$$

2(a). [5 Points] Find the relationship between continuous-time frequency Ω and discrete-time frequency ω which is defined by the bilinear transform, if

$$\begin{aligned}s &= j\Omega\\ z &= \mathrm{e}^{j\omega}\end{aligned}$$

2(b). [5 Points] Find the value T_d for which $\Omega \approx \omega_d$ at low frequencies, with $\omega_d \doteq \omega \cdot f_s$, at sampling rate f_s. **hint:** Use $\sin(x) \approx x$, $\cos(x) \approx 1$ for small x.

2(c). [5 Points] Find the value T_d for which $\Omega = \omega_d$ for some particular value Ω with sampling rate f_s. Your expression for T_d should be a function of Ω and f_s.

2(d). [10 Points] Populate the function `Bilinear` in the file `PluginProcessor.h`. This function takes as an argument the `double` value `'warpfreq'`. The argument specifies the frequency at which the responses for the continuous- and discrete-time systems should match. If the arguement has value zero, use the value for T_d you derived for the low-frequency bilinear approximation.

Your `Bilinear` function should use the continuous-time coefficients `b[]` and `a[]`, and write the discrete-time coeffients b_z, a_z as

$$\begin{aligned}
\texttt{targetCofs}[0] &= b_{z0}\\
\texttt{targetCofs}[1] &= b_{z1}\\
\texttt{targetCofs}[2] &= b_{z2}\\
\texttt{targetCofs}[3] &= a_{z1}\\
\texttt{targetCofs}[4] &= a_{z2}
\end{aligned}$$

The formatting of the coefficients assumes the difference equation has been normalized in the usual way, *i.e.* $a_{z0} = 1.0$. In other words,

$$y(n) = b_{z0}x(n) + b_{z1}x(n-1) + b_{z2}x(n-2) - a_{z1}y(n-1) - a_{z2}(n-2).$$

Problem 3. [10 Points]

3(a). [10 Points] Implement the function `SetSmoothingFactor` in the file `PluginProcessor.h`. This function generates the required constant for updating the filter coefficient values with leaky integration. The function takes the `double` argument `smoothingTime`, which is defined in seconds, and must take into account the sampling rate `samplingRate`. The calculated constant should be stored as `smoothingFactor`, and should be calulated using the Impulse Invariance method for modeling one-pole leaky integration with a time constant equal to the specified `smoothingTime`. To see how the constant is used to perform leaky integration, examine the function `SmoothCoefs`, which is directly above the function you are writing. Assume this function is called once per sample.

3(b). [15 Points, extra credit] Smoothing the raw filter coefficients can result in intermediate filters which do not closely resemble any of the modeled transfer functions. This is true because interpolating the filter coefficients is not equivalent to interpolating the parameters of the resonant peaking filter. An improvement can be made to the behavior by performing the leaky integration for smoothing on the parameters of the continuous time filter, and then performing the warped bilinear transform on the interpolated continuous-time design. This will add to the computational expense of the plugin, because multiple bilinear transforms must be calculated when the control is moved.

Implement a smoothing scheme that interpolates the parametric description of the filter. The brute-force method for doing this would be to re-calculate the bilinear transform every sample. Computation can be saved by creating a (small) set of intermediate filters which interpolate the parameters ω_0, Q, and γ, and then smoothing the raw filter coefficients on a sample-by-sample basis.

Problem 4. [10 Points]

Test the plug-in by playing the provided `.wav` file into it. Move the control the control back and forth and ensure that there are no discontinuities in the output.

Set the coefficient smoothing time to five seconds and listen to the filter changing as the control is moved between 0.0 and 1.0.

4(a). [10 Points] Record a one-second `.wav` file of the plug-in's response to an impulse, with the control set to 0.0, 0.5¡ and 1.0. Use Matlab to calculate the FFT of the three outputs, and turn in plots of the responses. Overlay the plots with the data from the measured ”wah” pedal at positions 0.0, 0.5, and 1.0. You may need to do some scaling on the plug-in output in order to get the plots to line up.