README file for LTSpice_opt program

LTSpice_opt is a Python program that uses an iterative optimization approach to design analog filters. It is designed to be used in conjunction with the popular circuit simulator LTspice. The program uses command-line control of LTspice to run hundreds or even thousands of frequency-response simulations in an attempt to match a user-defined target response. It works as follows;

- The user provides a target frequency and/or phase response in Python, and a circuit topology in LTspice with some initial component values.
- The user provides a list of which circuit instances the optimizer is allowed to vary. For component values that must maintain a fixed relationship to each other (for example, in fully differential filter structures), it is possible to also specify parameter values that will be part of the optimization process. Since component values can be linked to parameter names, this allows multiple component values to derive their value from a common parameter.
- The optimizer then iteratively adjusts those component and/or parameter values, using a 2-pass approach. In the first pass, a genetic algorithm called "differential evolution" is used to find a solution that is reasonably close to the optimum solution. Since genetic algorithms are not based on computing gradients, this first pass is more likely to avoid the common problem of converging to a local minimum. This is followed by a non-linear least-squares phase where the component values are fine-tuned to find the minimum distance between the desired target response and the computed response. The complete process may involve hundreds or thousands of spice simulations as the optimizer completes its work.
- Once the optimizer has finished, a new schematic is generated with the optimized component values. During the schematic generation process, each component value is quantized to a user-defined tolerance.

Why is this capability useful? Don't we already know how to design filters?

Traditional filter design uses standard circuit topologies such as Sallen-and-Key or multiple-feedback op-amp based active filters. In these cases, given a "standard" filter shape such a Butterworth or Chebychev, a high-order filter may be factored into 2nd-order sections, and an op-amp circuit can be used for each of those sections. This design procedure is quite straightforward and has not changed for many years. However, there are many cases where this approach fails;

- **The desired filter shape is not a traditional shape** such as Butterworth or Chebychev.
    - For example, the filter may need to compensate for some other part of the system that has a non-flat frequency response, while simultaneously attenuating other frequency regions.

- **The need to reduce power/area by combining multiple filter sections into a single op-amp circuit.**
    - This leads to non-conventional circuit topologies that have very messy closed-loop formulas, and it becomes very difficult to solve for the component values.

- **The application operates at frequencies where finite op-amp gain-bandwidth degrades the frequency response.**
    - Calculating the effects of finite gain-bandwidth on the frequency response is quite complicated, especially in cases where the gain/phase response deviates from the traditional single-pole model. By running optimizer simulations in LTSpice, the actual target op-amp may be included in the simulation. This yields a solution that inherently attempts to compensate for finite gain-bandwidth effects.

Python files included in the gitHub repo;

LTSpice_opt.py

- the main application

- the user must edit the 1st line of LTSpice_opt.py to import the setup file corresponding to the LTspice schematic to be optimized

Example setup files:

example1_setup.py, example2_setup.py, example2_diff_setup.py,hilbert_example_setup.py

Each of these files include 2 functions;

Set_target: for setting the target response

simControl: for setting schematic and system--specific variables

These functions can be copied and modified for any particular circuit. The following information needs to be provided;

- path to the LTSpice executable
- path to the LTspice circuit simulation directory
- file name of the LTspice circuit that will be optimized
- LTSpice variable that will be matched to the target
- list of instance names or parameter names in the schematic that the optimizer will adjust
- min values of the instances (or parameters) above
- max values of the instances (or parameters) above
- tolerance of the instances/parameters above in “E” format
- Match Mode, amplitude only, phase only, or both ampl+phase
- maxSpiceSims_de - the maximum number of spice sims used in the “differential evolution” phase. Typical values from 200 to 5000
- maxSpiceSims_lsq - the maximum number of spice sims used in the “least-squares” phase. Typical value from 200 to 2000.
- A target response (amplitude, phase, or both), calculated at the same frequencies used in the LTspice sim.

LTspice Files files included in gitHub repo

Example1.asc - a 3rd-order single-op-amp inverting lowpass filter. See the setup file for an example of how to enter a classic filter shape as the target response.

Example2.asc - a complicated single-op-amp filter including an LC stopband notch. See the setup file for an example of how to set a target response in the passband/stopband style.

Example2_diff.asc - same as above, but a differential version that shows how to optimize parameters to enforce component matching between the positive and negative halves of the circuit. See the setup file for an example of how to enter a mix of component instances and parameters.

Hilbert.asc - An example of an analog HIlbert transform filter, using two parallel banks of allpass filters whose outputs are 90 degrees apart over a wide frequency range.

Installation of the files

Place all the ".py" files in a directory where you will run Python.
Place all the .asc files in a directory where you will run LTspice.
The full-path location of the LTspice directory as well as the LTspice executable needs to be entered in the example setup files (see below).

Setting up the Python environment

The following must be installed in your Python environment;

numPy - included with most distributions
Matplotlib.pyplot - included with most distributions
PyLTSpice; from PyLTSpice · PyPI. This is used to read the LTspice .raw simulation files.

This code was developed using the popular Anaconda 3 environment, and run using a python command-line prompt (after setting up an env with the required modules installed).

Running the examples

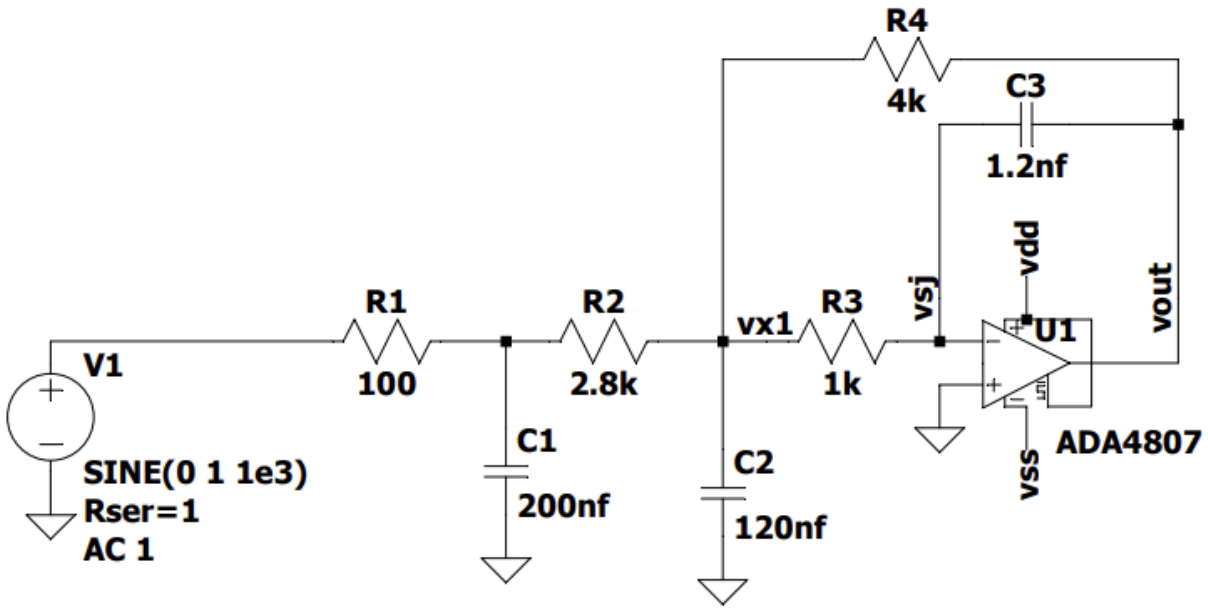
The setup files must first be modified to provide the correct paths to the LTspice executable and the LTspice schematic directory. The main application can then be started from a Python command line by running "Python -i LTspice_opt.py"
The first line of the main application is used to import one of the included example setup files; this line can be edited later to point to a user-defined circuit setup file.

Example circuits and results

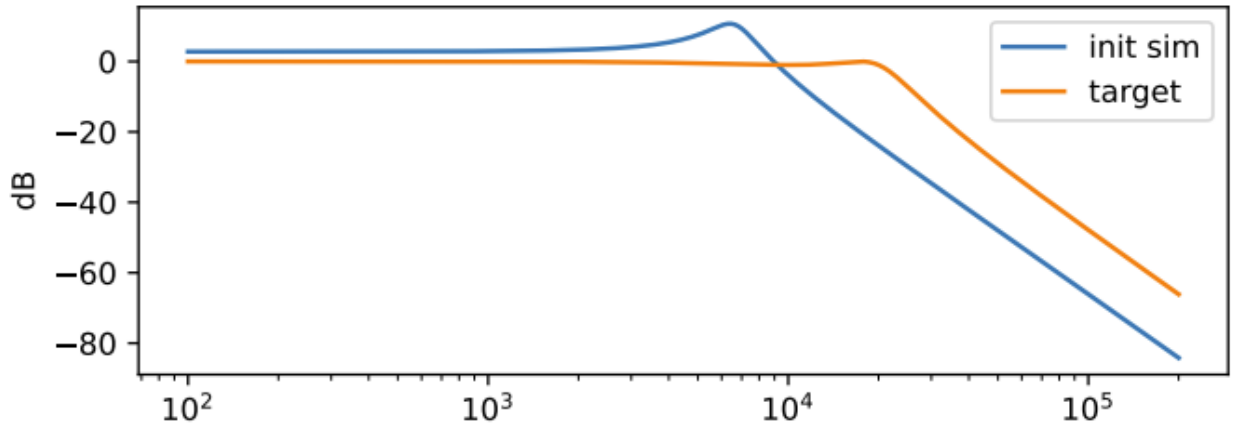
Example1.asc

The example1 circuit is a standard multi-feedback inverting lowpass filter, but with an extra passive RC added to the input. Since there is no buffer between the passive RC and the rest of the circuit, the equations become somewhat complicated, and an optimization approach can be used to make things easier. In this case, we suspect that we can match an ideal 3rd-order lowpass response, so we set our target response to a 1dB ripple chebychev1 type response, and use the Python function “cheby1” to generate the target.

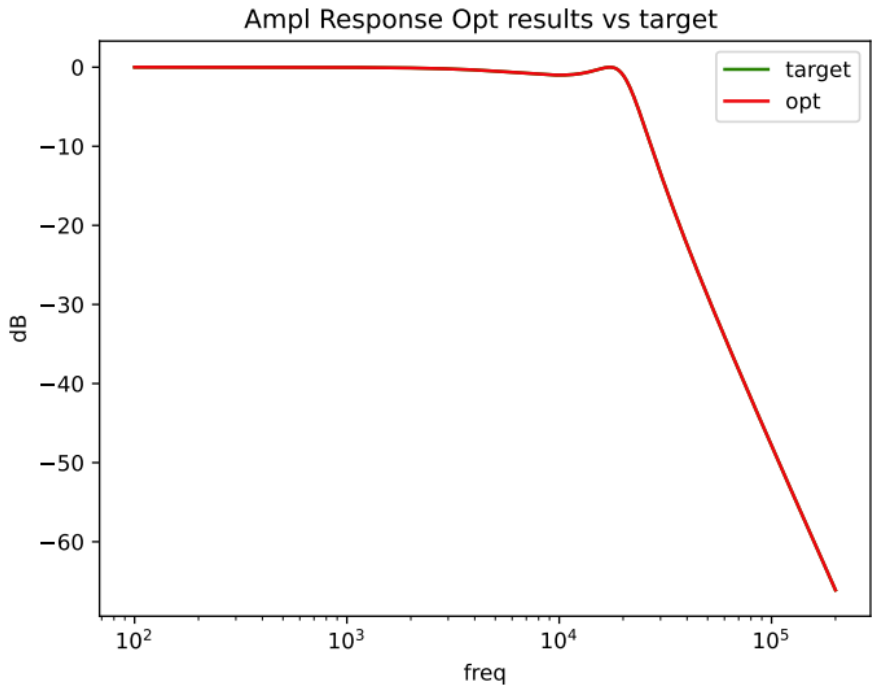
.ac oct 100 100 200e3



The pre-optimization response is show below;



The post-optimization response vs target is shown below. The curves are essentially on top of each other, showing that nearly ideal results are obtained.



Example2.asc

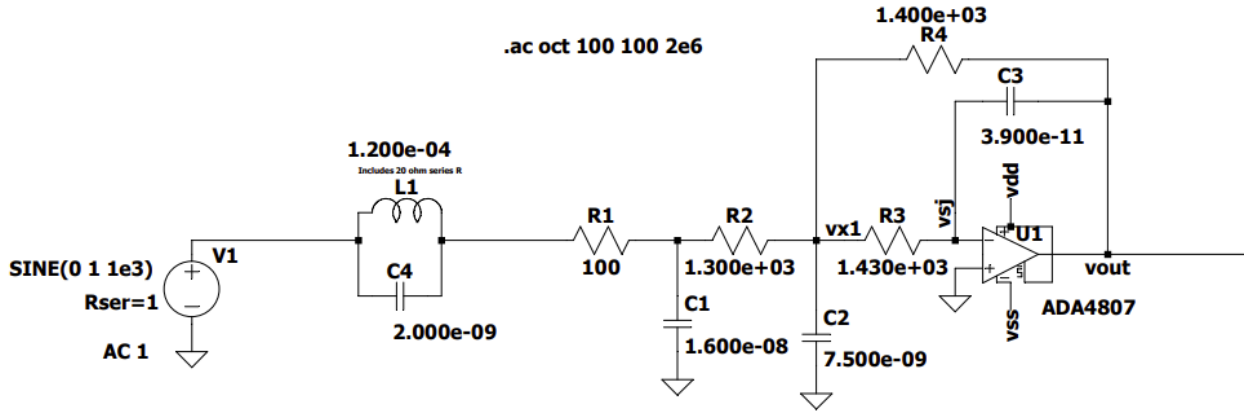
The circuit of “example2.asc” is shown below. This was an attempt to improve the stopband attenuation by inserting a passive parallel LC notch filter in series with the input. The true transfer function of this circuit would be extraordinarily

messy. We note that unlike the previous example, there is no standard target shape that we could use, so instead we use a passband/stopband/don't-care band approach, with the setup file set as follows;

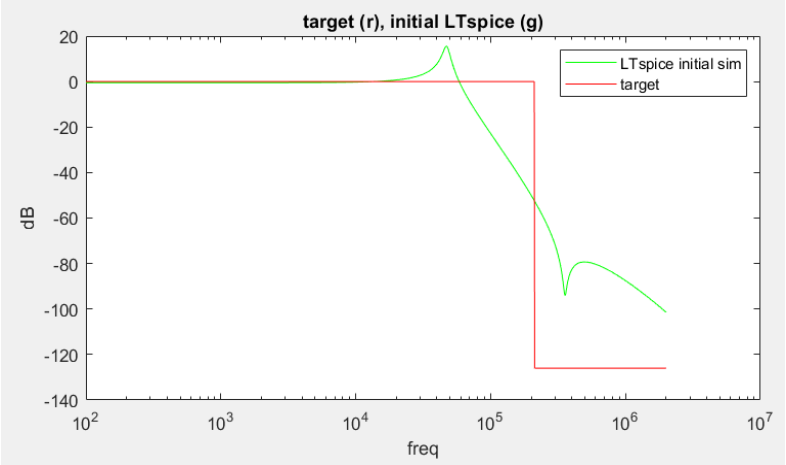
	Target	Error-weighting
Passband	1.0	1.0
Don't-care band	0.0	0.0
Stopband	0.0	10.0

The use of an error-weighting function allows us to make a "don't-care" region (by setting it to 0) and also allows us to set the relative ripple between the passband and stopband. Using this approach we obtain a lowpass shape that is quite sharp for a single op-amp filter.

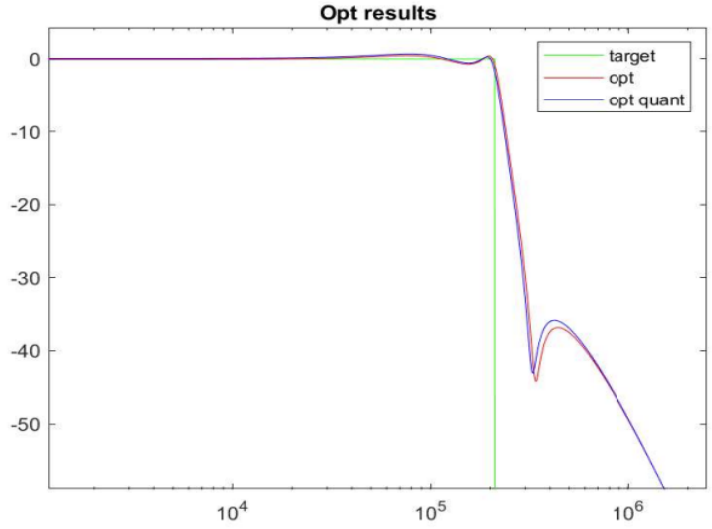
Schematic (post-optimization)



Pre-optimization simulation



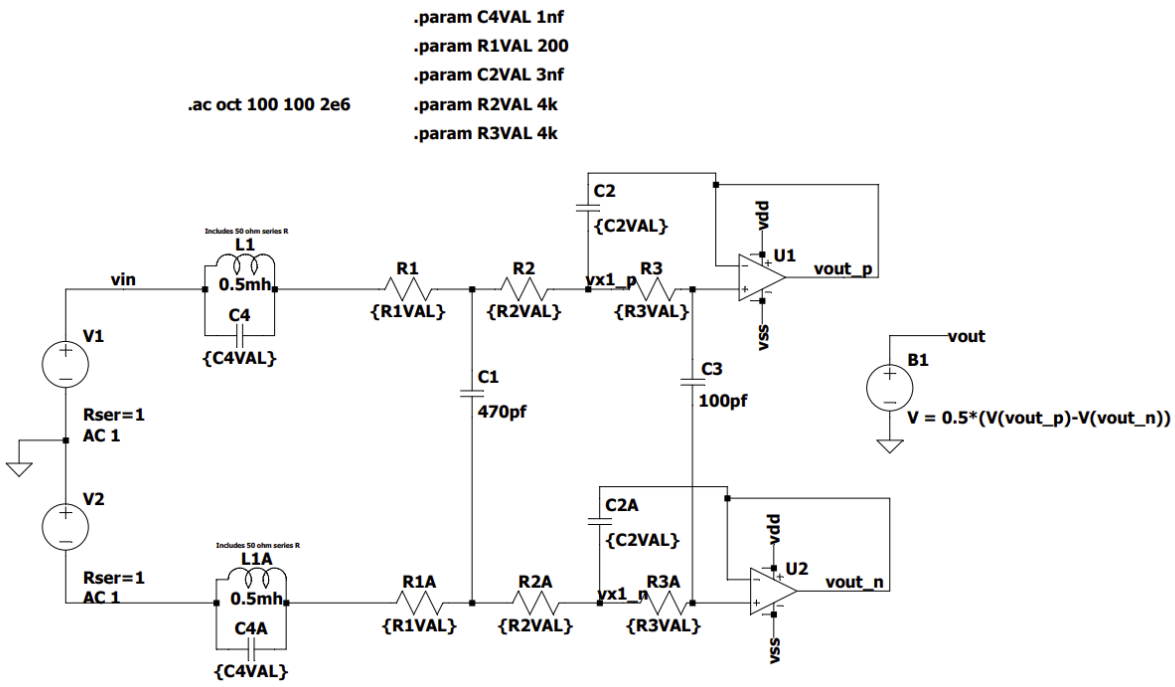
Post-optimization simulation



Example2_diff

A differential version of this same circuit is provided in “example2_diff”, shown below.

Schematic (pre optimization)



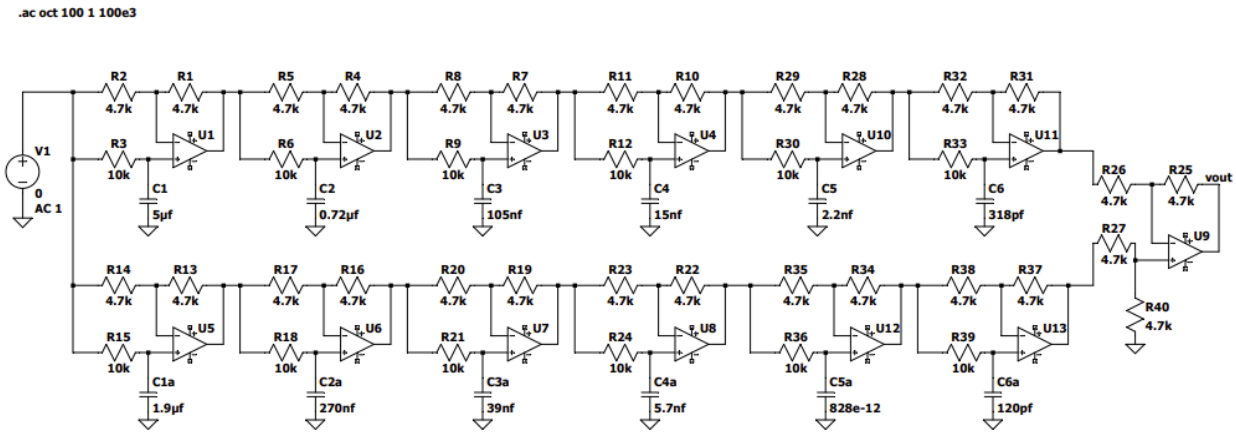
. This example shows how one can optimize parameters (listed on the schematic as .param simulation commands) rather than directly operating on the component values themselves. The component values can be set to some function of these parameters using curly braces as shown above. This allows multiple components to obtain their value from the same parameter. This is very useful for differential circuits where the positive and negative paths must use the same component values to ensure proper rejection of common-mode inputs. This concept can be extended further by entering scaling factors inside to curly braces to maintain a given ratio between pairs (or groups) of components (for example, '{2*R2VAL}').

To enable this feature, the user simply enters the parameter name rather than the instance name in the list of items to be optimized; the software will keep track of which names correspond to parameters and which names are component instance names.

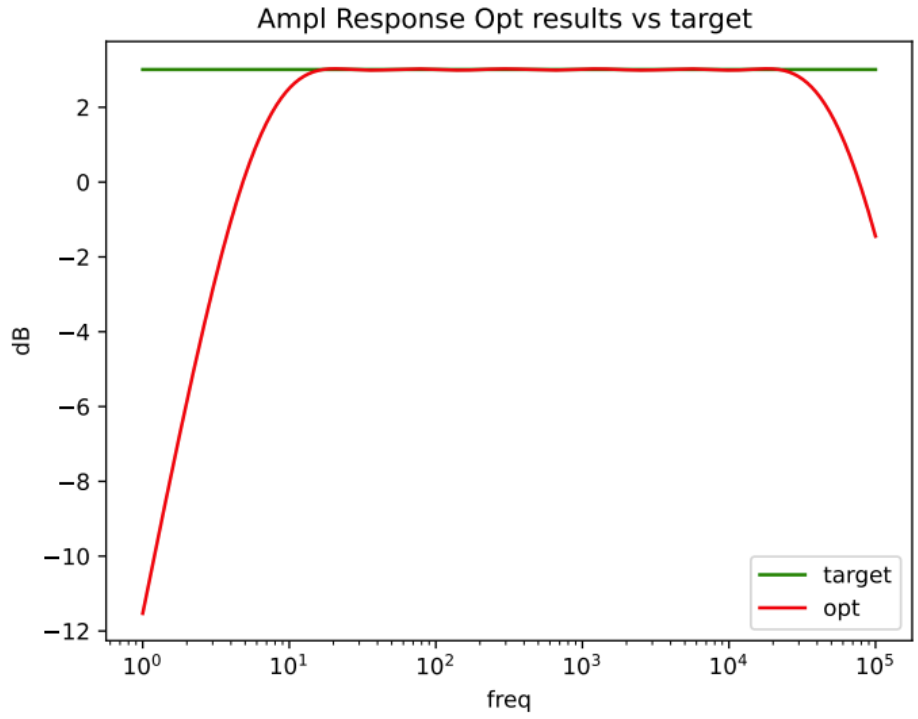
Hllbert_example

A common requirement for many systems is to create a complex signal given a single real-valued input. In digital systems, this can be accomplished using Hilbert FIR filters to create the imaginary part, but this approach is not possible for analog circuits. For analog circuits, one can obtain a 90 degree phase shift

over a finite frequency range by using two parallel chains of allpass filters. Each allpass filter chain exhibits a monotonically increasing phase shift as a function of frequency. If the poles and zeros of the two parallel branches are arranged correctly, then the difference in phase between the two outputs can be very close to 90 degrees. The circuit used is shown below;



Since there is no analytic solution to finding the allpass R-C values for each section in order to optimize the differential phase shift over some frequency range, an optimization approach is suitable.
For this case, we use the fact that the sum of a signal X(t) and it's quadrature component Y(t) should ideally have an amplitude of sqrt(2), so we sum the outputs of the two allpass chains and set a target amplitude response of +3dB over the audio band from 20-20KHz. After running the optimization (where 12 components are adjusted by the optimizer), we get the following result, which is very close to the ideal;



How to run your circuit;

1) Enter a schematic in LTSpice. Set the simulation command to do a frequency response sweep

example, '.ac oct 100 100 2e6'.

Run the simulation from within LTSpice to make sure there are no errors, and plot the frequency response of the output node. Note that the frequencies used in the LTSpice simulation are passed into the "setTarget" function as the variable "freqx". The target response must be computed at these same frequencies.
It's worth noting that if you use a log frequency grid in your simulation, the optimizer will pay equal "attention" to each octave, whereas if you use a linear frequency sweep in your simulation, the optimizer will pay equal attention to each Hz. It's worth thinking about which result you prefer; for audio circuits, a log-sweep is typically preferred.

2) Identify the component instance names and/or parameters that you want the optimizer to adjust. Note that making ALL the components adjustable is usually a bad strategy. For example, for most filters it is possible to scale the R's by some factor k, and also scale the C's by 1/k, without changing the frequency response. If all the R's and C's are adjustable, then the optimizer may drift off into an undesirable range of values, so it makes sense to pick one component that is not in the list of adjustable values, or alternatively restrict the range of one component to values that are acceptable.

3) Fill in the simControl function in the setup file. This file is heavily commented and it should be fairly clear how to enter your design.

The first step is to fill in the paths to your LTspice executable as well as the LTspice working directory (where you placed the .asc files).

Next, you need to fill in the information about which schematic instances and/or parameters will be adjusted by the optimizer. This information is stored in a list called **simControlOPtInstNames**. For each component or parameter listed, you must enter a min and max value, stored in lists **simControlMinVals** and **simControlMaxVals**. In most cases, the spread of min and max values should be fairly wide, to avoid limiting the optimizer; however, in some cases it makes sense for a particular component to have a more narrow allowable spread. For example, the min value of an input resistor may need to be large enough so it can be easily driven, but the maximum value may be limited by noise or bias current considerations.

In addition to the instance names, min values, and max values, the user should fill in the tolerance of the newly-adjusted components, stored in List **simControlInstTol**. This is specified in standard "E" format (number of values per decade), as defined in the comment section of the file. Note that the tolerance quantization is done "outside" the least-squares loop, as part of the schematic-generation process.

The example files can be copied and edited to minimize the chance of entry errors.

The “**matchMode**” parameter may be set to amplitude-only (1), phase-only (2), or both amplitude and phase (3).

The “**maxSpiceSims_de**” parameter controls how many iterations of the differential evolution method will be performed before exiting and passing the results on to the least-squares optimizer. For circuits with a large number of parameters (> 6, for example), this number may need to be fairly high

(1000-10000) in order to get good results. This may require a fairly long time to complete, allowing the user to drink several cups of coffee and contemplate life in general.
The “**maxSpiceSims_lsq**” parameter controls how many iterations of the least-squares algorithm will be executed before returning the optimized result. Typical values range from 500-5000, allowing even more coffee to be consumed.

4) Edit the setTarget function in the setup file. This function contains a frequency vector called "**freqx**" that will be imported from the main program, and will contain all the frequencies that are used in the LTSpice sim. You must generate a vector called "**target**" with the dimensions as freqx. "Target" should contain the desired magnitude response (note, NOT in dB) of your filter at the corresponding frequencies in freqx. In the case of matching phase-only, the target should be set to the unwrapped phase in radians. For both magnitude and phase matching, target should be the concatenation of amplitude and phase targets.

Also in the setTarget function is a vector called "**errWeights**". This vector controls how much to weight the error at the corresponding frequencies in the freqx vector. It is initialized to all 1's, which may be adequate for many designs. However, when the filter design contains some combination of passbands, stopbands, and transition bands, it may be useful to set the weighting to 0 in the transition bands, and to some larger number in the stopbands. A rule of thumb is that the ratio of linear passband ripple to linear stopband ripple should equal the ratio of the stopband weights to the passband weights.
If the goal is to design a classic filter shape such as Butterworth or Chebyshev, then the built-in sciPy signal-processing functions may be used to set the target response. An example of this technique is shown in the circuit “example1.asc’ with the corresponding setup file “example1_setup.py”.
In cases where the target response values are not on an x grid that is identical to the LTspice frequency sweep vector, it is possible to use the numpy “interp” function to interpolate the target response to the LTspice frequency grid.

Note that the sciPy final optimizer is a least-squares optimizer, and therefore cannot be expected to return an equiripple design. However, the user can experiment with running the algorithm multiple times, and for each new run, adjust the weighting factors in regions where the ripple is too high. This can eventually result in a nearly equiripple design.

5) Run the main program LTSpice_opt.. See the requirements for setting up your Python environment above.

If everything is set correctly, you will see a series of updates in the Python command window, which tracks the algorithm progress. For each iteration of the genetic algorithm and least-squares algorithm, the component values are displayed, along with the rms error in the frequency response versus the target. Depending on many factors, the optimizer may run for only a few minutes, or for much longer if the design is complicated.

In addition to the updated component values, the window displays the cumulative number of simulations.

When the algorithm has finished, a plot is displayed of the target response, the optimized response with arbitrary-precision components, and the optimized response with components that were quantized according to user input as described earlier. A new schematic is also generated with the quantized optimized component values, with a schematic name the same as the original but with "_opt" appended to the name. It is advisable to open this schematic and run the simulation from LTspice to make sure that everything has worked as expected. If the schematic contains both parameters and components that are to be adjusted, both the parameters and component values will reflect the optimized values.

If the quantization of component values to their specified tolerance results in excessive error, a useful strategy is to take the worst-tolerance components (for example, 5% capacitors) and remove them from the list of instances to be optimized, and then re-run the optimizer based on the new '_opt' schematic. This will cause the more precise components (for example, 1% resistors) to adjust their value slightly to compensate for the effect of the 5% quantization of the capacitor values.

Caveats

There are some circuits that have difficulty finding a DC operating point before running the AC sim. In these cases, not only is the optimizer very slow, but there may be an issue where a new DC solution is found for each pass through the optimizer, and this can affect the AC response in ways that are not linear. This behavior is common when using op-amps with non-zero input bias currents (typically bipolar op-amps). If you encounter such a case, it is wise to use ideal op-amps with no input bias currents. In many cases the user can set the

open-loop gain and the bandwidth parameters of the ideal op-amp to match the datasheet of the real op-amp, and this will give sufficient accuracy.

Acknowledgements

This development was inspired by a program written more than 40 years ago by **Mark Davis**, who was my co-worker at the time at dbx Inc. Mark is an MIT PHd graduate, and has made extensive contributions to the field of audio during the course of his career. His original program ran in a DOS command window and was 100% text-based. Mark used it to design crossover filters for loudspeakers (where the target response came from a speaker measurement program), while others used it to design non-standard op-amp circuits. I hope this program proves to be as useful as the original!

License