CS110.A Laboratory Work: Week 4

This laboratory work aims at understanding what the “box” type filter does to sine wave input - we have seen in the videos that, while initially there is some distortion (“transient response”) eventually the output ends up being a sine wave; possibly with some delay, and a different amplitude.

I have written the following MATLAB function (you need to put it in a file called filter_power.m):

```
function power = filter_power(filter, f)
    k = 0:999;
    x = sin(2*pi*k*f/1000);

    pin = x * x';

    y = conv(filter, x);
    y = y(1:1000);

    pout = y * y';

    power = pout/pin;

end
```

The idea here is this: We define the power of a discrete-time signal as the sums of the squares of its samples (strictly speaking, this is proportional to the energy, but we keep our sample size at 1000 samples, so it does not matter much).

This function aims at computing what a given filter (which should be a row vector) does to the power of a 1000-sample sine wave of frequency f. Here, the units of f are a bit funny, having f=1 corresponds to having one full wave in the sample range, having f=2 corresponds to having two full waves, so on and so forth.

The variable “pin” stands for “power in”. You can verify that this always evaluates to 500 as long as $0 < f < 1000$. But this is the power of the input signal.

We do the convolution, and snip back to 1000 samples.

The variable “pout” is the power output. This really depends on the “filter” and “f” arguments.

The function returns the ratio of the output power to input power… Which is one measure of what the filter does to a sine wave.

1. Type the function into MATLAB.

2. Run:

```
>> filter_power([0 1 1], 5)

ans =

   3.998993399698108
```

This is expected, since we saw using [0 1 1] doubles the amplitude. Doubling the amplitude quadruples the power. So yeah.

3. Run:

```
>> filter_power(ones(1, 50)/50, 1)

ans =

   0.991481605380204
```

This is a 50-sample wide, but “normalized” (divided by 50) square filter. This is applied to an f=1 signal (period T=1000). Most of the power (99.1%) gets through. Note that some of the “power” gets clipped at the end, but we are fine.

4. Run:

```
>> filter_power(ones(1, 50)/50, 5)

ans =

   0.805097182375960
```

Hmm, f=5, T=200. 20% of power is gone now.

5. Run:

```
>> filter_power(ones(1, 50)/50, 10)

ans =

   0.400340368047931
```

Oh, power has gone below half.

6. Run:

```
>> filter_power(ones(1, 50)/50, 8.748)

ans =

   0.500057591191731
```

A-ha! The half-power point. Well, almost. The frequency argument can take fractional values, there is actually not a problem with that. This frequency corresponds to a period of about 114.3 samples.

7. Now, the idea is to plot how this power thing changes with the frequency involved. We have just looked at specific values. Now we want to plot this stuff!

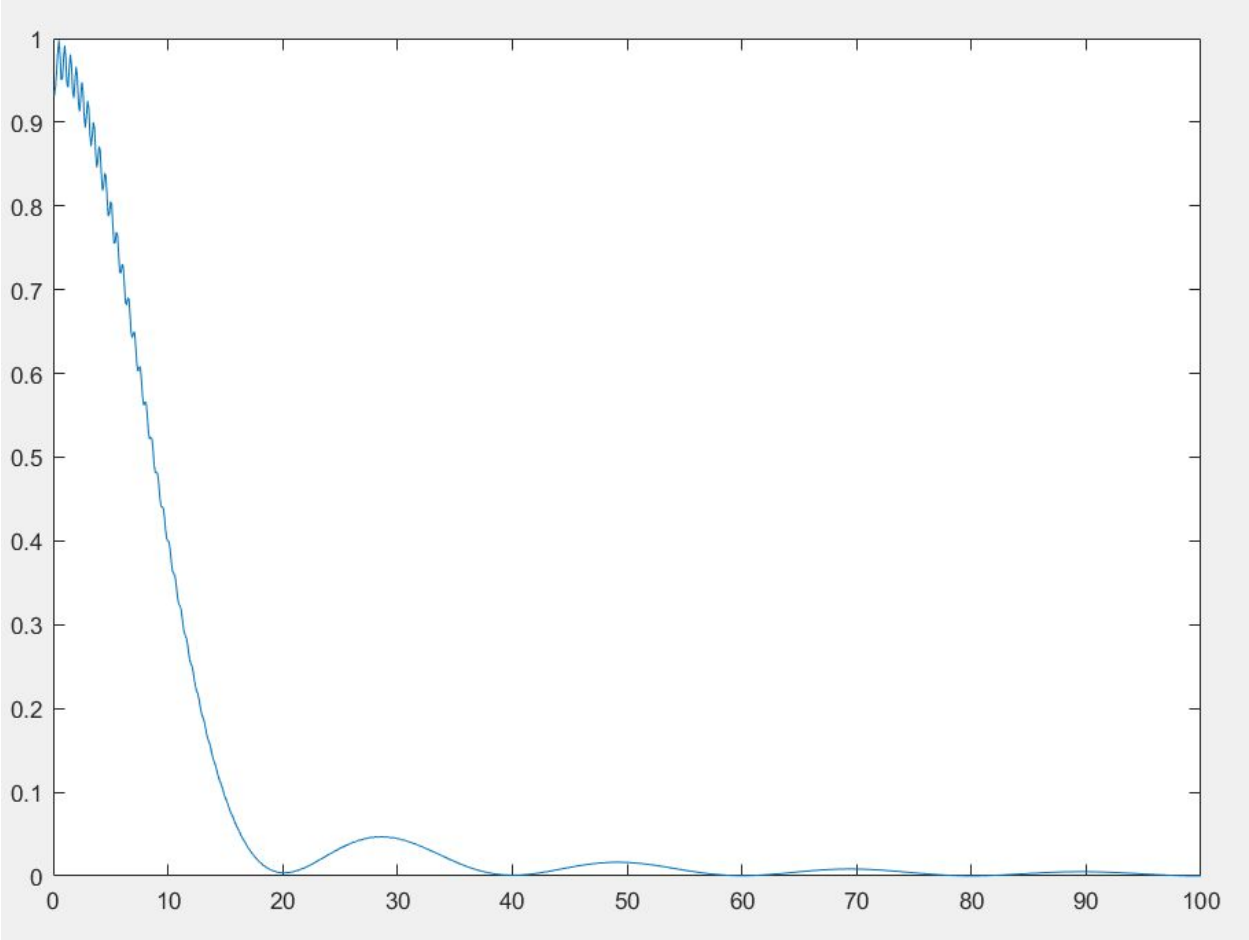
Enter this in a function file called power_plot.m:

```
function power_plot(filter)

    power = [];
    frequencies = 0.1:0.1:100;

    for f = frequencies
        power = [power filter_power(filter, f)];
    end

    plot(frequencies, power);

end
```

8. Now, run this with our 50-sample wide square filter:

```
>> power_plot(ones(1, 50)/50);
```

The result should look like this:



At f=20, nothing passes through. This should be expected, since then the period is exactly 50, and averaging a sine wave over a full period always gives a zero. Zeros also happen at 40, 60, 80… But stuff does survive in between. Filtering is not perfect business.