

Supplement to Chapter 4 of *The Science of Digital Media* – Digital Audio Representation

Worksheet – Digital Audio > Root Mean Square Amplitude¹

Introduction:

The **root-mean-square amplitude (RMS amplitude)** of a signal relates to the average amplitude of a sound wave over a given period. This period can either be the entire wave or a portion that you have selected. The formula for r , the RMS amplitude of a signal, is as follows:

Let N be the number of samples in an audio signal. Let x_i is the amplitude of the i^{th} sample. Then the **root-mean-square amplitude**, r , is defined as

$$r = \sqrt{\frac{1}{N} \sum_{i=1}^N x_i^2}$$

Equation 1

(Different audio processing programs use different terms for what we have called "RMS amplitude." Adobe Audition uses the term "RMS power"; Sony Sound Forge uses "RMS level." We prefer the more generic and descriptive term "RMS amplitude.")

RMS amplitude can be expressed in terms of sample values or in **decibels-full-scale (dBFS)**. If r is the RMS on the scale of sample values and n is the bit depth of the audio file, then the RMS can be converted to dBFS with the following equation:

$$dBFS = 20 \log_{10} \left(\frac{r}{2^{n-1}} \right)$$

Equation 2

Tips for Using MATLAB:

We'll implement RMS amplitude using MATLAB's programming language, which resembles C in many respects.

To create a function in MATLAB, go to File\New\m-file. (You can also use an external text editor like NotePad or WordPad.) In MATLAB, the function file is called an m-file, which contains the code to run a function that can later be called from the MATLAB command window. The following syntax is used to declare a function *test* that accepts one input argument (*in*) and one output argument (*out*).

¹This material is based on work supported by the National Science Foundation under Grant No. DUE-0340969. This worksheet was written by Todd Martin and Jennifer Burg (burg@wfu.edu).

function out=test(in)

In MATLAB, the input argument is not strongly typed; it can be either a single numerical value or an array, and the function declaration would be the same. In our function, the function receives an array of *doubles* as input, representing the amplitude values of the signal in the time domain.

The body of the function is written in the m-file. Each line (except the declaration of a flow control statement or a function) must be followed by a semicolon, like in C. In our function we will first make sure that there is *exactly* one input argument when the function is called. To do this, we will use the *nargin* keyword, as follows:

```
if(nargin ~= 1)  
    error('Incorrect number of input arguments.');  
end
```

This is the general format for all *if* statements in an m-file. We will next use a *for* loop to cycle through all of the elements of the input array. A basic *for* loop in MATLAB is implemented as follows:

```
N=length(in);  
runningSum=0;  
for(i=1:1:N)  
    runningSum = runningSum + (in(i) * in(i));  
end
```

This sum will then need to be divided by the *N* value, and the square root will need to be taken. The square root can be implemented with the MATLAB *sqrt* function within the m-file. There is not any *end* command at the end of the function itself. When the entire function is complete, save the text file as *RMS.m* and return to the MATLAB command window.

In order to execute the function, you need to include the folder in which you saved *RMS.m* in the MATLAB path list. MATLAB searches the folders in this list when you call a function in the command window. The path list can be altered by clicking File/Set Path/Add Folder. (Or you can change the current directory to the one that contains the m-file.)

The function can be executed from the command window as follows:

```
>> r=RMS(signal);
```

After this command is executed, the variable *r* contains the RMS amplitude. You don't need to include the conversion to dBFS in the RMS program; we'll look at both values and manually convert between them.

Exercise 1

Implement the RMS function in an m-file saved as *RMS.m*. Use a single input argument named *signal* and an output argument named *r*.

```
>> r=RMS(signal);
```

Exercise 2

Now let's compute the average amplitude of a simple "signal" that we will define manually. This signal will contain 11 samples, starting with a value of 100 and increasing by 10 until reaching a final sample value of 200. This is just a simple, contrived example to explain the basic concepts and test the correctness of your program.

Store this signal in the variable *signal1*. To do this, the following command can be used:

```
>> signal1=[100:10:200];
```

Calculate the RMS of this signal using the function that you created and stored in *RMS.m*. Store the result in a variable called *r1*. Does this answer make sense, based on the "averaging" and square root that the formula includes? Using the formula for RMS, calculate the answer by hand and compare with the result given by *RMS.m*.

Exercise 3

Now let's calculate the RMS of a more larger signal – a sine wave with a frequency of 100 Hz. First define the period – or window – of the signal using a time variable *t* and use a window of 0 to 1.999 seconds with 1000 samples/second – a total of 2000 samples. This can be done in the following MATLAB command:

```
>> t=0:0.001:1.999;
```

Next, define the 100 Hz sine wave and assign it to the variable *signal2* with the following function:

```
>> signal2=sin(100*2*pi*t);
```

The result of this assignment is an array of sample values for the signal. Now, calculate the RMS of the wave stored in *signal2* and store the result in *r2*.

Exercise 4 (*Any digital audio processing program that calculates RMS amplitude can be used*)

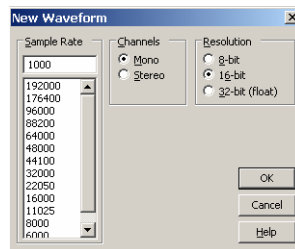
You've already calculated the RMS of a 100 Hz wave with a MATLAB program. This result is in terms of amplitude, but some audio processing programs – e.g., Adobe Audition – display RMS in decibels-full-scale (dBFS). Let's

convert to dBFS, allowing us to compare the result of our MATLAB calculation with what we see in Audition's statistics window.

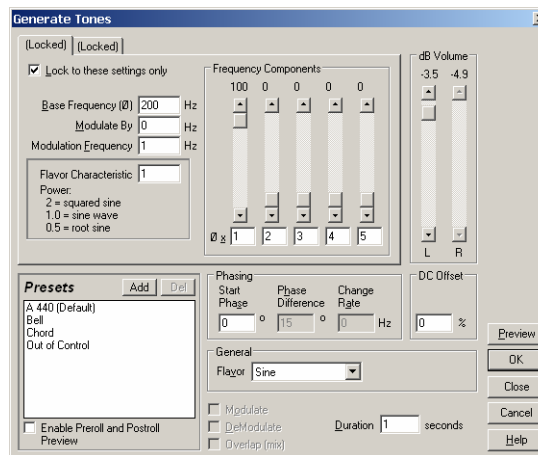
There are two ways to obtain the signal for this exercise. If you have Adobe Audition or another equivalent sound processing program, follow the instructions below. If you don't have these programs, skip the following steps and follow the second set of instructions.

Instructions for Adobe Audition

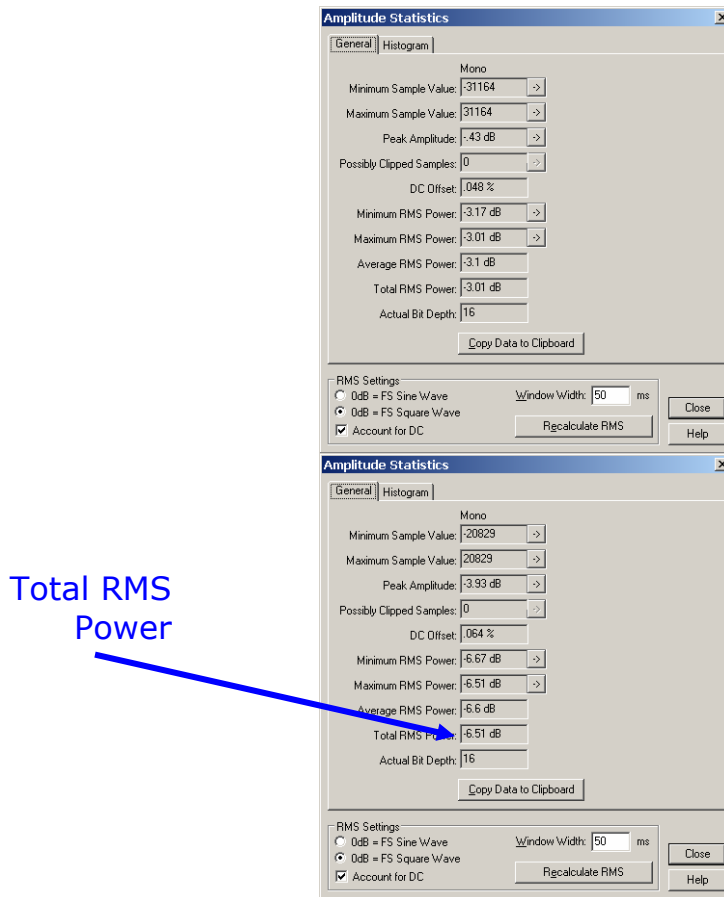
Create a 200 Hz wave in Audition by opening the program and clicking File/New. In the New Waveform menu, enter a sampling rate of 1000 Hz and choose Mono and 16-bit resolution.



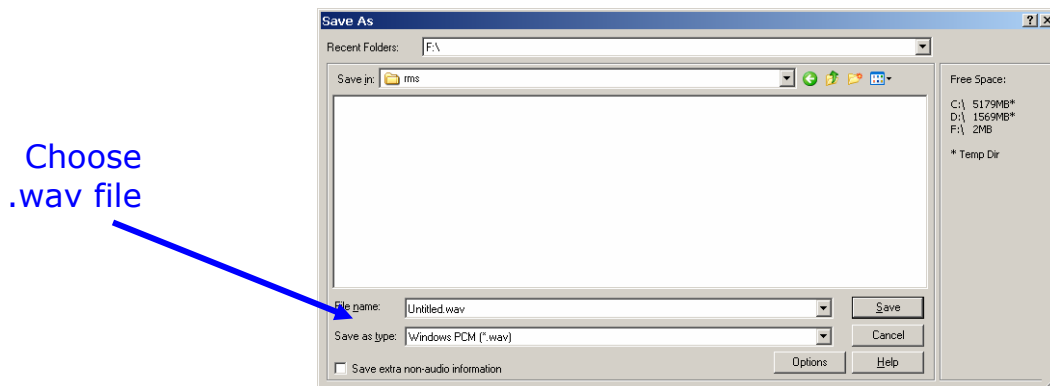
You should now have a blank waveform window. Click Generate/Tones, and at the Generate Tones menu choose a base frequency of 200 Hz. Make sure that the dB volume meter is set to 0 dB and set the duration to 2 seconds.



You should now have a waveform on your screen. To find the RMS amplitude of the wave according to Audition, click Analyze/Statistics. We're interested in "Total RMS Power" value. Find this value and take note of it.

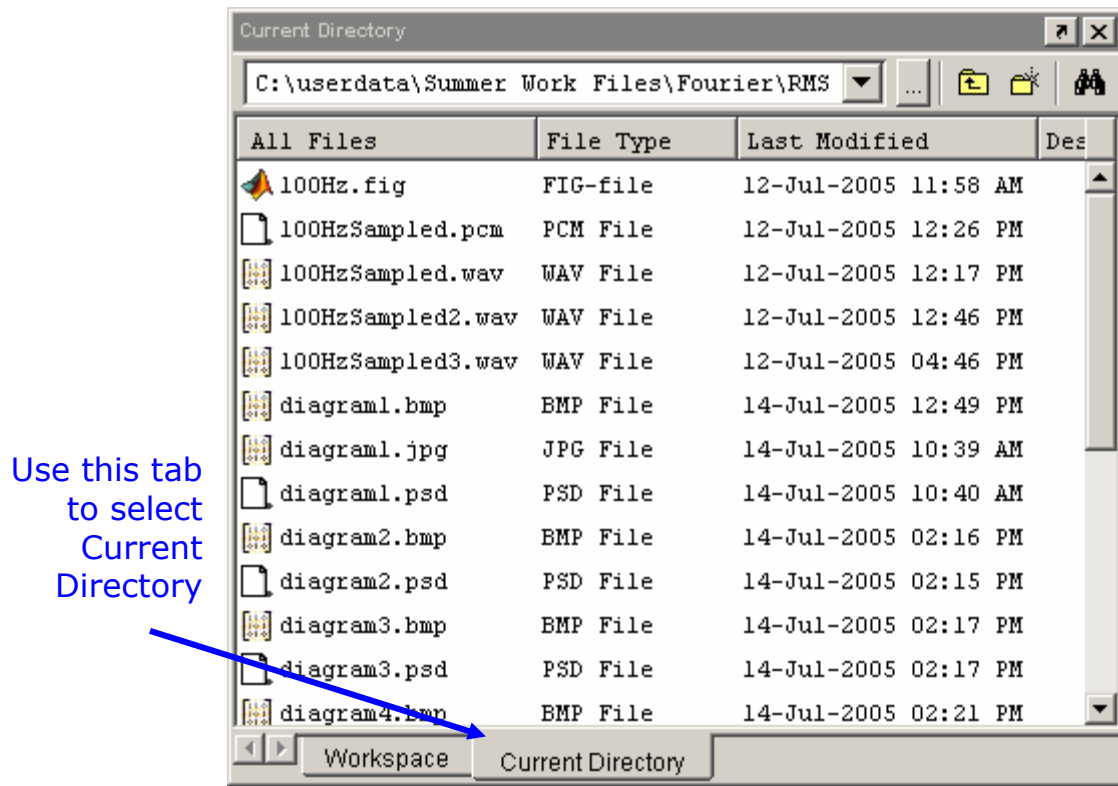


Now, export the wave as a *.wav* file. Do this by clicking File/Save As and choosing the file type to be Windows PCM (**.wav*). Choose the folder in which you would like to save the file and save the file as *signal3.wav*.



The next step is to import *signal3.wav* into MATLAB and store it in a variable called *signal3*. This variable will point to an array of sample values for the wave. To import into MATLAB, first set the current path in MATLAB to the file path of the folder that contains the file. The

current path can be set by clicking on the "Current Path" tab underneath the Workspace window.



Once this is set, use the following command to load the file.

```
>> signal3=wavread('signal3.wav');
```

Skip the following section for those without Adobe Audition and continue with the exercise.

Instructions without Adobe Audition

Create a 200 Hz sine wave using the example given in Example 3 and store it in the variable *signal3*. The same *t* values can be used.

Through either of the above sets of instructions, you have created a 200 Hz signal and stored it in a variable called *signal3* in MATLAB. This signal must now be adjusted in order to correspond to the form used by Audition to calculate its RMS.

When Audition calculates the RMS of the waveform in *signal3*, it uses a bit depth of 16. This means that the sample values vary between $-32,768$ and $32,767$ – a range of $2^{16}=65,536$ possible values. The values in *signal3* in MATLAB, however, vary between -1 and 1 . To adjust *signal3* to match the bit depth used by Audition, each of the sample values must be multiplied by $32,768$. To do this, use the following command:

```
>> signal3Adjusted=signal3*32768;
```

Now, use this *signal3Adjusted* to calculate the RMS. Use the program that you wrote in Exercise 1. Store the result in *r3*.

The result of this computation is in terms of amplitude. In order to convert to dBFS, use Equation 2 with *r3* as the *r* value in the equation. Also, assume a bit depth of 16 bits to mirror the 16-bit depth that Audition uses to calculate the RMS power of this wave. This means that *n* must be set to 16. In MATLAB, the *log10* command can be used for the logarithm. Store the result of this calculation in the variable *r3dBFS*.

If you have Adobe Audition and created the 200 Hz wave for Exercise 4 in this program, compare the value in *r3dBFS* and the "Total RMS Power" value shown in the Audition statistics window. (If you have Sound Forge, look in the Statistics window. Other audio processing programs have similar features for RMS.)