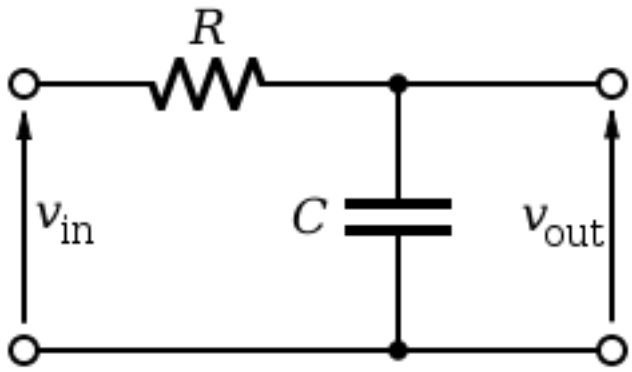
MP 3

MP3 is due on Saturday, October 12, by 8pm in your Subversion repository.

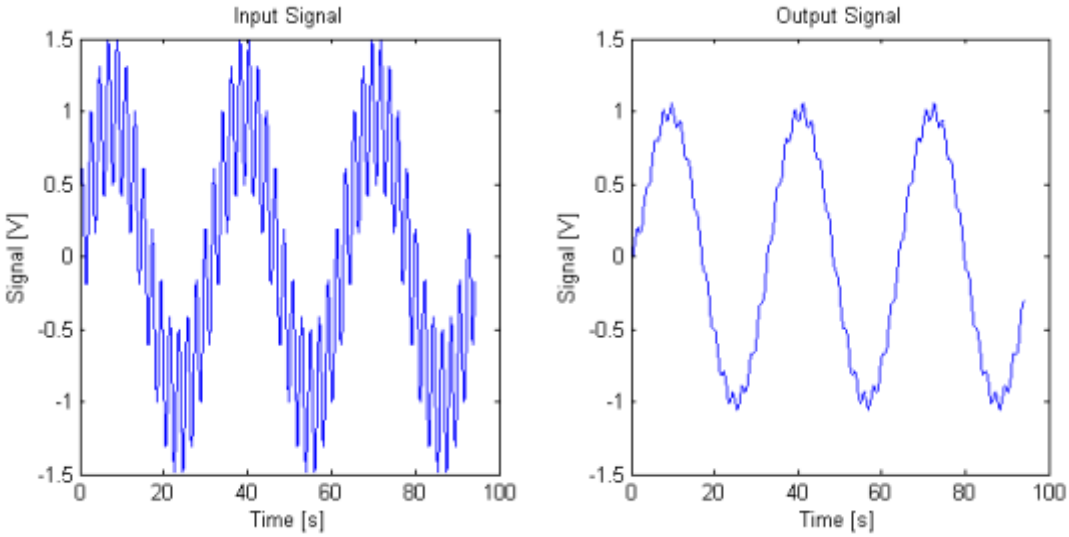
Lowpass filter

Your task this week is to implement a low-pass filter in software. An ideal low-pass filter takes an input signal, completely attenuates its components above the cutoff frequency, and passes everything below the cutoff frequency as its output signal. Low pass filtering is common in signal processing when a signal below a specific frequency is desired, for example, in eliminating background noise from music.

Shown below is an example of a first-order low-pass filter. Filters like this are not ideal, which means that they will not completely attenuate signals above the cutoff frequency. However, if the signal has a high-frequency component that lies well above the cutoff frequency, it will be greatly attenuated.



Shown below is a plot of input signal that is passed through a low-pass filter. The input signal has high-frequency component, but you can see in the output signal that this component has been attenuated by the filter.



In the programming lab we worked on the code for computing a function value at some discrete points. You should reuse that code in this MP.

This is **not** a collaborative MP. You must work on it alone.

The Pieces

In the MP3 folder, you will find the following files:

main.c – This file contains the main function which reads in the inputs. You will modify this file where it is indicated in order to implement the low pass filter.

mp3-compare – This is a program that enables you to compare output produced by your program with the correct results.

There are also some test input files. Each of these test files contains *omega1, omega2, omegac,* and *method number*, respectively. Read on to understand what these parameters mean.

Details

In this MP, you will implement a second-order low-pass filter using the finite difference method. The finite difference method is a useful mathematical method that is used to numerically solve differential equations. Numerical techniques become very handy when there is no analytical solution for a given equation. In the finite difference method, all continuous signals are discretized as

$$V(t) = V(n\Delta t) = V[n] \text{ where } n = 0, 1, 2, \ldots, N-1$$

Here, Δt is the time step and *N* is the total number of time steps. $V[n]$ is just a convenient way to express *V* at time step $n\Delta t$. Another approximation used in the finite difference method is the discretization of derivatives. Derivatives are approximated by finite differences of discrete signals:

$\frac{d}{dt}V(t) \approx \frac{V[n+1]-V[n]}{\Delta t}$.

The differential equation for the second-order low-pass filter is given by

$$V_{out}(t) + \frac{\sqrt{2}}{\omega_c}\frac{d}{dt}V_{out}(t) + \frac{1}{\omega_c^2}\frac{d^2}{dt^2}V_{out}(t) = V_{in}(t)$$

where ω_c is the cutoff frequency. For the MP, we will consider an input signal given by $V_{in}(t) = sin(\omega_1 t) + 0.5sin(\omega_2 t)$. You can play with other input signals later. In the test input files, we set the frequencies so that $\omega_1 \leq \omega_c \leq \omega_2$. Our goal is to numerically solve the above differential equation and find the approximate solution to $V_{out}[n]$. Using the finite difference method, the differential equation is discretized as follows.

$$V_{out}[n+1] = \frac{1}{\frac{1}{\sqrt{2}\Delta t\omega_c} + \frac{1}{\Delta t^2\omega_c^2}}\left[\left(\frac{2}{\Delta t^2\omega_c^2} - 1\right)V_{out}[n] + \left(\frac{1}{\sqrt{2}\Delta t\omega_c} - \frac{1}{\Delta t^2\omega_c^2}\right)V_{out}[n-1] + \sin(\omega_1 n\Delta t) + 0.5\sin(\omega_2 n\Delta t)\right]$$

This is the time-stepping equation that you have to evaluate for this MP. For every time step *n*, you use the values of the input and output signals at step *n* or before to calculate the output signal at time step *n*+1. You may assume that $v_{out}[n] = 0 \text{ for } n < 0$.

Once you implement this, you need to output your calculated values of $v_{out}[n] \text{ for } n = 0, 1, 2..N-1$ in the following format:

```
printf("%lf\n", variable_name);
```

Here, *variable_name* corresponds to $V_{out}[n]$

Building and Testing

To compile your code, use the following command:

```
gcc -Wall -g main.c -o mp3 -lm
```

To run your program, type:

```
./mp3 testone.txt
```

Here are a few lines of output for **testone.txt** model:

```
0.000000
```

```
0.000000
0.001791
0.006746
0.015719
0.029005
0.046334
0.066936
0.089636
0.112993
0.135468
0.155583
0.172089
0.184101
0.191193
0.193451
0.191472
...
```

You have two more input test files other than *testone.txt*. Each of these text files contains four numbers which correspond to *omega1, omega2, omegac,* and *method number,* respectively. The *method numbers* are set to 1 for these test files. 1 means you are running the finite difference method. If you are doing the challenge, write 2 in that place to pick your Runge-Kutta function. You may also create your own test files to test your code.

To compare your output to the solution, type:

```
./mp3 testone.txt > myoutone
./mp3-compare testone.txt myoutone
```

Your output file, *myoutone,* contains output of your program. *mp3-compare* takes your output and calculates how close you are to the "correct" output signal given the inputs in *testone.txt*. Make sure that *myoutone* was calculated using *testone.txt* from your mp3 code. Otherwise, you might see a huge error.

You may observe that your results are not exactly the same as the ones posted above. That's expected, see this discussion for details.

Formats, Comments, and Introduction Paragraphs

In order to get full points on Comments and Style, and also Introduction Paragraphs, you shall carefuly arrange your code in order to makes it easy to read and understand. You may follow these code as an example, or defer to the Coding Conventions page:

Sample Coding

```
/* This program calculates the area of a circle with given radius. */   <---
The purpose of Program
```

```
/* It accepts a integer from stdin (terminal), and puts a float number to
stdout (terminal). */   <--- How it is used

#include <math.h>

int main()
{
    int r;   /* radius */
    float s; /* area */

    /* Input Radius */
    printf("Input the radius: ");
    scanf("%d", &r);

    /* Calculate Area */
    s = M_PI * r * r;

    /* Output Area */
    print("%f", s);

    return 0;
}
```

Typically, good and easy to understand code includes blanks which seperates blocks, indentations and alignments keep blocks neat, and comments.

Challenges

Runge-Kutta method (+15%)

As a challenge problem, you will solve the same differential equation using the different numerical technique called the Runge-Kutta method. This method breaks the second-order differential equations down into two first-order differential equations.

$$\text{Let } V_{out1}(t) = V_{out}(t) \text{ and } V_{out2}(t) = \frac{d}{dt}V_{out}(t).$$

Then, we get a set of two coupled equations:

$$\frac{d}{dt}V_{out1}(t) = V_{out2}(t)$$
$$\frac{d}{dt}V_{out2}(t) = \omega_c^2 V_{in}(t) - \omega_c^2 V_{out1}(t) - \omega_c\sqrt{2}V_{out2}(t)$$

Applying the Runge-Kutta method to these two equations yield the following set of discrete equations:

$$V_{out1}[n+1] = V_{out1}[n] + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where $k_1 = V_{out2}[n], \quad k_2 = V_{out2}[n] + \frac{k_1 \Delta t}{2}, \quad k_3 = V_{out2}[n] + \frac{k_2 \Delta t}{2}, \quad k_4 = V_{out2}[n] + k_3 \Delta t$

$$V_{out2}[n+1] = V_{out2}[n] + \frac{\Delta t}{6}(m_1 + 2m_2 + 2m_3 + m_4)$$

where $m_1 = \omega_c^2(\sin(\omega_1 n \Delta t) + 0.5\sin(\omega_2 n \Delta t)) - \omega_c^2 V_{out1}[n] - \omega_c \sqrt{2} V_{out2}[n], \quad m_2 = m_1\left(1 + \frac{\Delta t}{2}\right), \quad m_3 = m_1 + \frac{m_2 \Delta t}{2}, \quad m_4 = m_1 + m_3 \Delta t$

You may assume that $V_{out1}[n] = 0 \text{ and } V_{out2}[n] = 0 \text{ for } n < 0$. You also need to output the values of $V_{out}[n] \text{ for } n = 0, 1, 2, ..., N-1$ in the same format as you have done in this MP.

mp3-compare only implements the first method, it does not implement the second method and should not be used to compare it.

Grading Rubric

- *Functionality (90%)*
  - The program outputs the correct results for various inputs.
- *Style, comments, clarity, and write-up* (10%)
  - Comments and Style - 5%
  - Intro Paragraph - 5%
- *Challenge (15%)*
- If your code does not compile you will get 0.