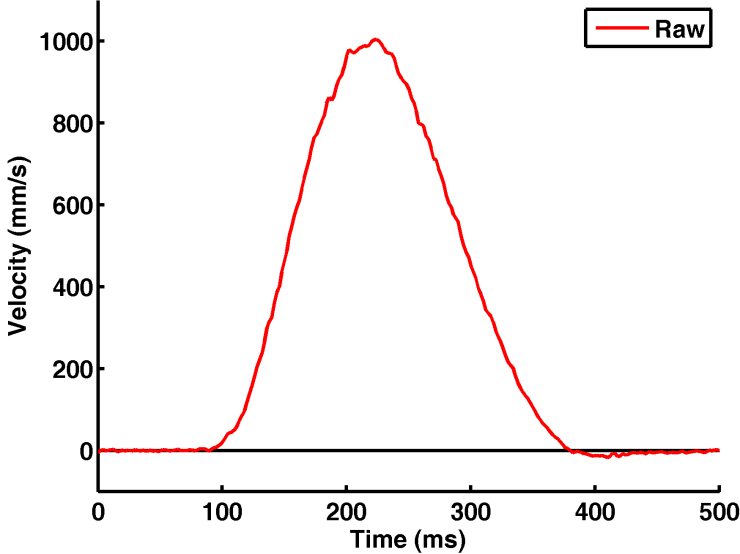
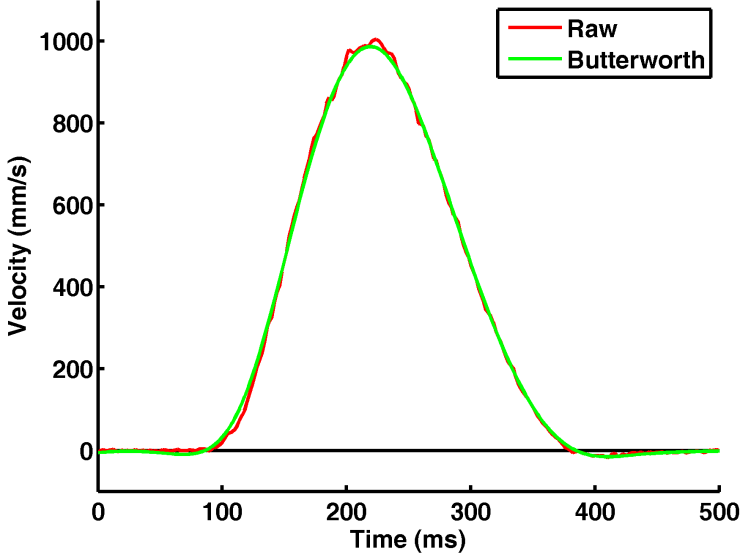
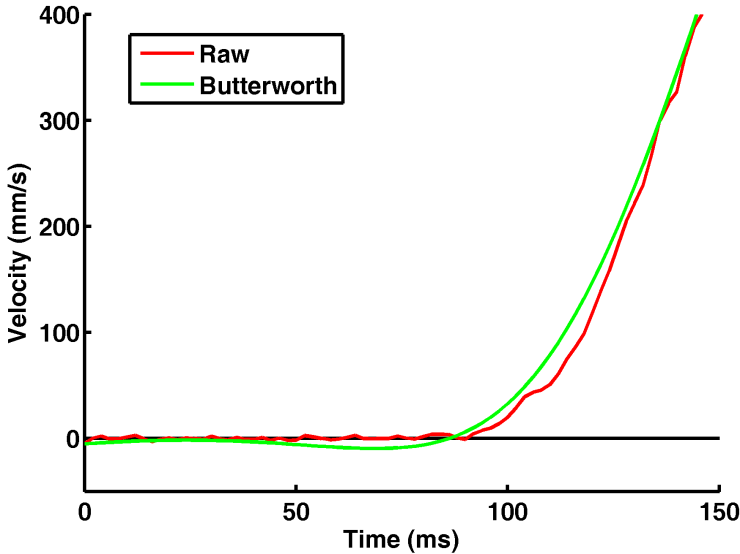
**Solving the underdamped response of a 10 Hz low pass Butterworth filter**

巴特沃斯滤波器的使用在神经力学研究中非常普遍。但是，它可能会扭曲您的数据。 第一个失真是单通滤波器将时间事件在时间上向前“移动”（向右），这称为相移。 解决方案是对数据进行两次过滤，首先是正向过滤，然后是反向过滤。第一遍会将事件暂时向右移动，第二遍会将事件移回其开始的位置； 这些称为双通滤波器。

下图演示了第二种失真。第一个图是弹道臂运动的原始速度剖面。这些数据是在 500 Hz 频率下收集的。

下图显示了通过 10 Hz 低通巴特沃斯滤波器（双通道，每通道 2 阶，这使其成为 4 阶滤波器）后的速度剖面。 您可以看到这减少了原始数据中的噪声或“抖动”。



但让我们仔细看看速度曲线的开始部分。 请注意，过滤后的数据低于原始数据（从 40 到 80 毫秒），然后又高于原始数据（从 80 到 140 毫秒）。 当运动可以向前或向后方向开始时，这尤其成问题。 您可能会想知道参与者是否在向前移动之前做了向后的小移动，或者这是否是过滤的产物！

每当数据发生快速转变时（例如在弹道运动的开始和结束时），巴特沃斯滤波器就会产生这些下冲和过冲。 这个问题可以通过临界阻尼滤波器来解决。 相比之下，巴特沃斯滤波器是欠阻尼滤波器。 许多人推荐巴特沃斯滤波器而不是临界阻尼滤波器。 David Winter 教授和 Aftab Patla 教授认为，“由于脉冲或阶跃输入在人类运动数据中很少见，因此首选巴特沃斯滤波器”（运动科学的信号处理和线性系统，第 31 页）。 对于某些人类运动来说确实如此，但对于弹道运动绝对不是这样。

在本文的其余部分中，我将介绍如何在 Matlab 中实现临界阻尼滤波器，并将比较巴特沃斯滤波器和临界阻尼滤波器。

## Matlab 中的巴特沃斯和临界阻尼滤波器

有关如何实现巴特沃斯和临界阻尼滤波器的更多详细信息，请参阅 Winter 和 Patla 所著的《运动科学的信号处理和线性系统》中的第 1 章和第 2 章。 这本书很难找到，幸运的是，有一篇由 Robertson 和 Dowling 撰写的文章，名为“Design and responses of Butterworth and critically damped digital filters (2003, Journal of Electromyography and Kinesiology)”，可以在线获取。

我们的目标是制作双通四阶（每通二阶）的 10 Hz 低通滤波器。

### the code for a Butterworth filter：

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear all;

clc;

sampling\_frequency = 500; % Hz

cutoff\_frequency = 10; % Hz, low pass

filter\_passes = 2; % dual pass

% Maintain the cutoff frequency by adjusting for multiple passes (10 Hz becomes 12.4650 Hz).

c\_butter = 1 / (((2^(1/filter\_passes))-1)^(1/4));

f\_adjusted\_butter = cutoff\_frequency \* c\_butter;

% Use Matlab commands to create a Butterworth filter.

[B, A] = butter(2, f\_adjusted\_butter / (sampling\_frequency / 2), ‘low’);

% You could now have Matlab filter the data with the following command.

% filtered\_data\_array = filtfilt(B\_low, A\_low, raw\_data\_array);

% Instead of relying on Matlab, let’s code our own Butterworth filter.

% First we need to calculate the five coefficients (ao, a1, a2, b1, b2) for the following equation of a second order filter.

% yn = aoxn + a1xn-1 + a2xn-2 + b1yn-1 + b2yn-2

% x is the data we will filter, and y is the data after filtering.

% n is the current data point, n-1 is the previous data point, and n-2 is the one before that.

% It is a recursive formula in that each filtered data point involves 3 points of raw data and 2 points of filtered data.

% Important: These coefficients are different than the three B and three A ones calculated with the butter command, but I will show you how to convert from one to the other.

w\_adjusted\_butter = tan(pi\*f\_adjusted\_butter/sampling\_frequency);

k1\_butter = sqrt(2) \* w\_adjusted\_butter;

k2\_butter = w\_adjusted\_butter^2;

a0\_butter = k2\_butter / (1 + k1\_butter + k2\_butter);

a2\_butter = a0\_butter;

a1\_butter = 2 \* a0\_butter;

b1\_butter = 2 \* a0\_butter \* (1/k2\_butter – 1);

b2\_butter = 1 – (a0\_butter + a1\_butter + a2\_butter + b1\_butter);

% We can do two sanity checks on our coefficients.

% First, ao + a1 + a2 + b1 + b2 should equal 1.

% This means that the filter has a gain of one in the band-pass region.

display(a0\_butter + a1\_butter + a2\_butter + b1\_butter + b2\_butter);

% Second we can convert our coefficients into B and A arrays to ensure they match the Matlab butter command.

% More details on the mathematical equations are available on the Cookbook formulae for audio EQ biquad filter coefficients.

B\_manual\_butter = [ a0\_butter a1\_butter a2\_butter ];

A\_manual\_butter = [ 1 -b1\_butter -b2\_butter ];

display(B);

display(B\_manual\_butter);

display(A);

display(A\_manual\_butter);

% Open a column of data to filter – this is the velocity data from above.

% You can download the data here, but you will need to change the extension from .pdf to .mat .

t = open(‘data\_raw\_array.mat’);

data\_raw\_array = t.data\_raw\_array;

% Use this instead for a step response.

%data\_raw\_array = [ zeros(1,500) ones(1,500) ];

data\_raw\_copy\_array = data\_raw\_array;

% Now filter the data.

data\_butter\_array = ones(size(data\_raw\_array)) .\* NaN;

% First pass in the forward direction.

for x = 1:size(data\_raw\_array,1)

if (x >= 3)

data\_butter\_array(x) = a0\_butter\*data\_raw\_array(x) + a1\_butter\*data\_raw\_array(x-1) + a2\_butter\*data\_raw\_array(x-2) + b1\_butter\*data\_butter\_array(x-1) + b2\_butter\*data\_butter\_array(x-2);

elseif (x == 2)

data\_butter\_array(x) = a0\_butter\*data\_raw\_array(x) + a1\_butter\*data\_raw\_array(x-1) + b1\_butter\*data\_butter\_array(x-1);

else

data\_butter\_array(x) = a0\_butter\*data\_raw\_array(x);

end

end

% Flip the data for the second pass.

data\_raw\_array = data\_butter\_array(end:-1:1); % this is now the once filtered data

data\_butter\_array = ones(size(data\_raw\_array)) .\* NaN;

% Second pass.

for x = 1:size(data\_raw\_array,1)

if (x >= 3)

data\_butter\_array(x) = a0\_butter\*data\_raw\_array(x) + a1\_butter\*data\_raw\_array(x-1) + a2\_butter\*data\_raw\_array(x-2) + b1\_butter\*data\_butter\_array(x-1) + b2\_butter\*data\_butter\_array(x-2);

elseif (x == 2)

data\_butter\_array(x) = a0\_butter\*data\_raw\_array(x) + a1\_butter\*data\_raw\_array(x-1) + b1\_butter\*data\_butter\_array(x-1);

else

data\_butter\_array(x) = a0\_butter\*data\_raw\_array(x);

end

end

% Flip the data back.

data\_butter\_array = data\_butter\_array(end:-1:1);

% Restore the raw array.

data\_raw\_array = data\_raw\_copy\_array;

% Plot the results.

figure;

hold on;

plot(data\_raw\_array, ‘r’);

plot(data\_butter\_array, ‘g’);

legend(‘Raw’, ‘Butterworth’);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

### the code for critically damped filter：

c\_critical = 1 / (((2^(1/(2\*filter\_passes)))-1)^(1/2));

f\_adjusted\_critical = cutoff\_frequency \* c\_critical; % 10 Hz becomes 22.9896 Hz

w\_adjusted\_critical = tan(pi\*f\_adjusted\_critical/sampling\_frequency);

k1\_critical = 2 \* w\_adjusted\_critical;

k2\_critical = w\_adjusted\_critical^2;

a0\_critical = k2\_critical / (1 + k1\_critical + k2\_critical);

a2\_critical = a0\_critical;

a1\_critical = 2 \* a0\_critical;

b1\_critical = 2 \* a0\_critical \* (1/k2\_critical – 1);

b2\_critical = 1 – (a0\_critical + a1\_critical + a2\_critical + b1\_critical);

display(a0\_critical + a1\_critical + a2\_critical + b1\_critical + b2\_critical);

% Convert the coefficients to B and A. There is no Matlab command to compare these to, but we will use them later.

B\_manual\_critical = [ a0\_critical a1\_critical a2\_critical ];

A\_manual\_critical = [ 1 -b1\_critical -b2\_critical ];

% Filter the data.

data\_critical\_array = ones(size(data\_raw\_array)) .\* NaN;

% First pass in the forward direction.

for x = 1:size(data\_raw\_array,1)

if (x >= 3)

data\_critical\_array(x) = a0\_critical\*data\_raw\_array(x) + a1\_critical\*data\_raw\_array(x-1) + a2\_critical\*data\_raw\_array(x-2) + b1\_critical\*data\_critical\_array(x-1) + b2\_critical\*data\_critical\_array(x-2);

elseif (x == 2)

data\_critical\_array(x) = a0\_critical\*data\_raw\_array(x) + a1\_critical\*data\_raw\_array(x-1) + b1\_critical\*data\_critical\_array(x-1);

else

data\_critical\_array(x) = a0\_critical\*data\_raw\_array(x);

end

end

% Flip the data for the second pass.

data\_raw\_array = data\_critical\_array(end:-1:1); % this is now the once filtered data

data\_critical\_array = ones(size(data\_raw\_array)) .\* NaN;

% Second pass.

for x = 1:size(data\_raw\_array,1)

if (x >= 3)

data\_critical\_array(x) = a0\_critical\*data\_raw\_array(x) + a1\_critical\*data\_raw\_array(x-1) + a2\_critical\*data\_raw\_array(x-2) + b1\_critical\*data\_critical\_array(x-1) + b2\_critical\*data\_critical\_array(x-2);

elseif (x == 2)

data\_critical\_array(x) = a0\_critical\*data\_raw\_array(x) + a1\_critical\*data\_raw\_array(x-1) + b1\_critical\*data\_critical\_array(x-1);

else

data\_critical\_array(x) = a0\_critical\*data\_raw\_array(x);

end

end

% Flip the data back.

data\_critical\_array = data\_critical\_array(end:-1:1);

% Restore the raw array.

data\_raw\_array = data\_raw\_copy\_array;

% Plot the results.

figure;

hold on;

plot(data\_raw\_array, ‘r’);

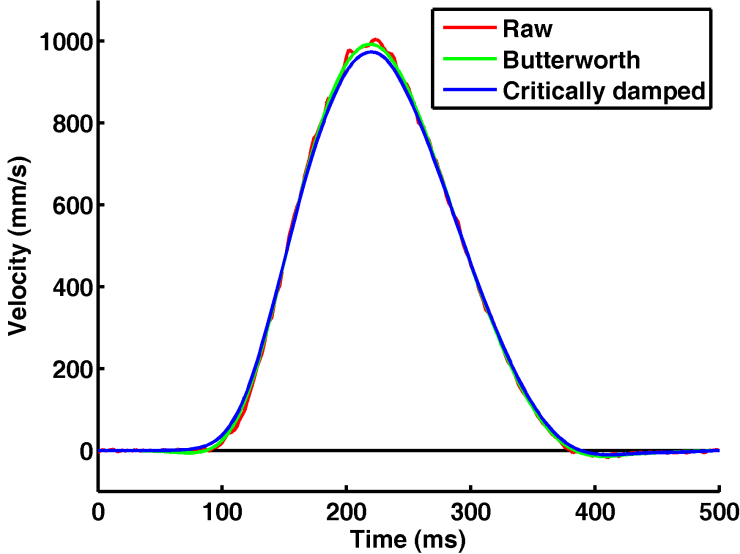
plot(data\_butter\_array, ‘g’);

plot(data\_critical\_array, ‘b’);

legend(‘Raw’, ‘Butterworth’, ‘Critically damped’);

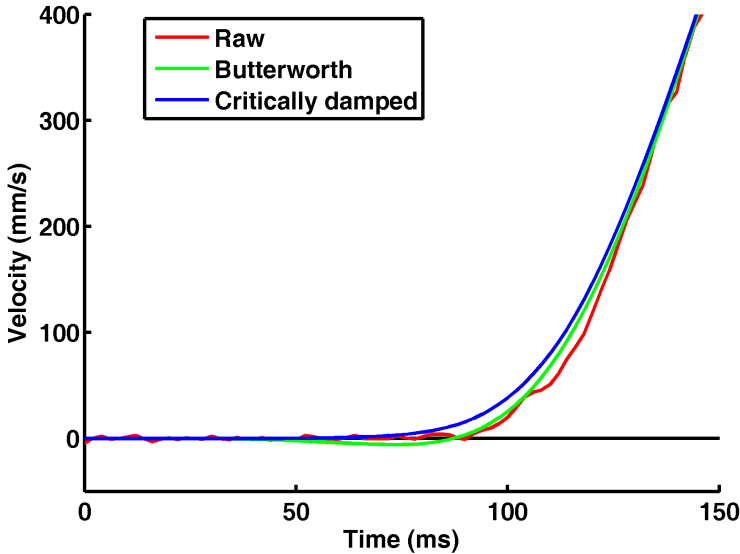
### Comparing Butterworth and critically damped filters

所有这些代码的输出如下图所示。

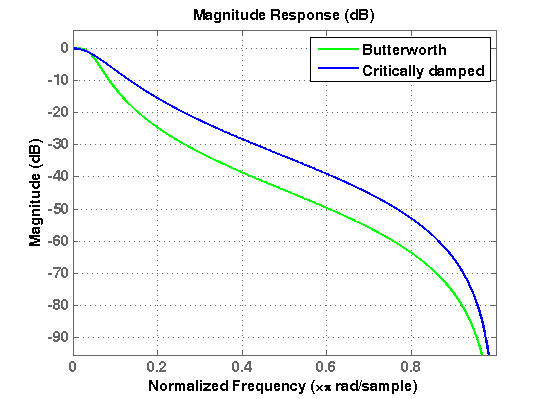


请注意，临界阻尼数据解决了巴特沃斯数据的负速度问题（下一个图在运动开始时放大）。 然而，临界阻尼数据与原始数据不同； 与原始数据相比，临界阻尼数据中的速度较早增加，然后低于峰值速度。

看来临界阻尼滤波器的改进并非没有缺点。 巴特沃斯滤波器的响应速度更快，它会更接近原始数据，但当存在快速转变时，它可能会朝相反的方向发展。 临界阻尼滤波器的响应速度较慢。 它不会朝相反的方向发展，但对快速转变的反应较慢。



这些滤波器的差异通过绘制每个滤波器的幅度响应来显示。

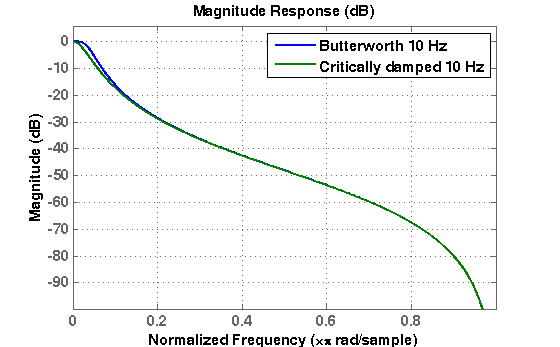


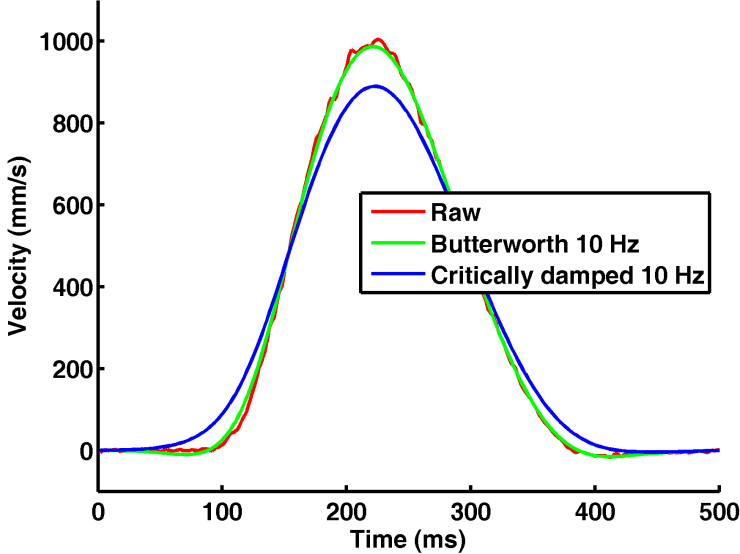
您可以使用以下代码在 Matlab 中亲自尝试一下：

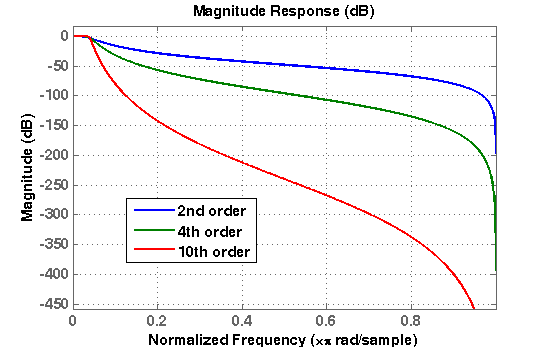
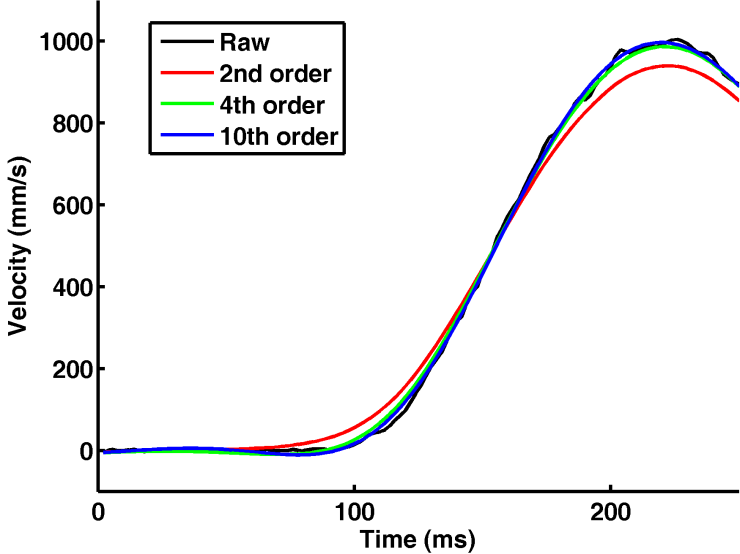
fvtool(B\_manual\_butter, A\_manual\_butter, B\_manual\_ritic, A\_manual\_ritic);

legend('巴特沃斯', '临界阻尼');

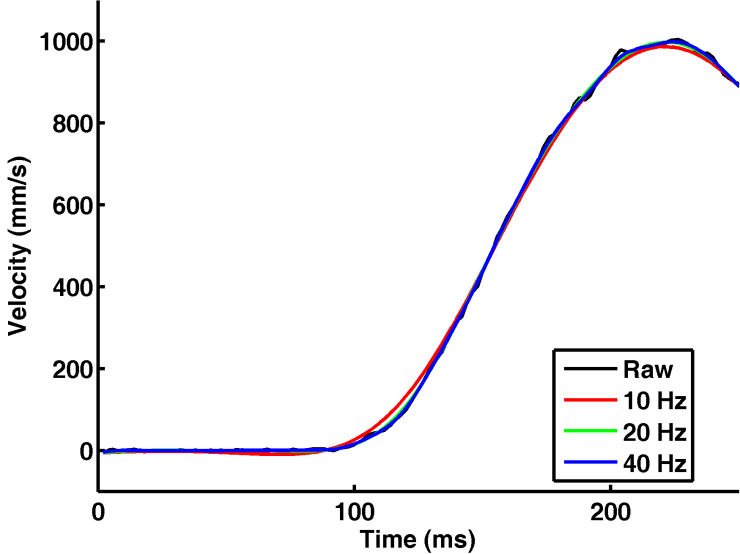
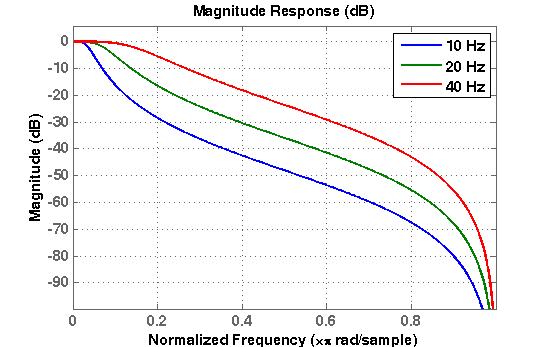
## Update (June 24):

我与 Jean-Sébastien Blouin 教授讨论了这些过滤器，他鼓励我尝试更多的东西。 首先，我尝试将巴特沃斯滤波器和临界阻尼滤波器设置为 10 Hz，以便进行更好的比较。 幅度响应相似，但看看低频的微小差异如何影响数据。 临界阻尼滤波器落后于原始数据太多，并且巴特沃斯滤波器在运动开始和结束时仍然有振铃。



然后，我比较了三个不同阶数的 10 Hz 巴特沃斯滤波器：二阶、四阶和十阶。 幅度响应截然不同，但速度分布相似。 请注意，2 阶滤波器太慢，4 阶滤波器更好，但存在振铃，而 10 阶滤波器则振铃变得更糟。

最后，我尝试了具有不同低通频率的四阶巴特沃斯滤波器：10、20 和 40 Hz。 幅度响应的主要区别在于具有较高截止频率的滤波器保留更多的较低频率（正如它们应该的那样）。 好消息是 20 和 40 Hz 滤波器在运动开始和结束时没有振铃！ 我认为问题在于 10 Hz 滤波器消除了一些对数据（而不仅仅是噪声）很重要的频率。 这种弹道运动的频率高于 10 赫兹是有道理的。



这里重要的一点是您可以使用巴特沃斯滤波器（不需要使用临界阻尼滤波器）。 但是，在设置截止频率时，必须考虑数据中的频率。 我已经使用 10 Hz 太久了。 弹道运动需要更高的截止值； 20 Hz 对于该数据效果很好。 请注意不要将滤波器设置得太高，否则 60 Hz 噪声可能会潜入您的数据中。

总的收获是了解您的数据！ 查看原始数据，并查看过滤后的数据。 如果速度曲线的开始和结束处出现振铃，请尝试增加滤波器的频率。