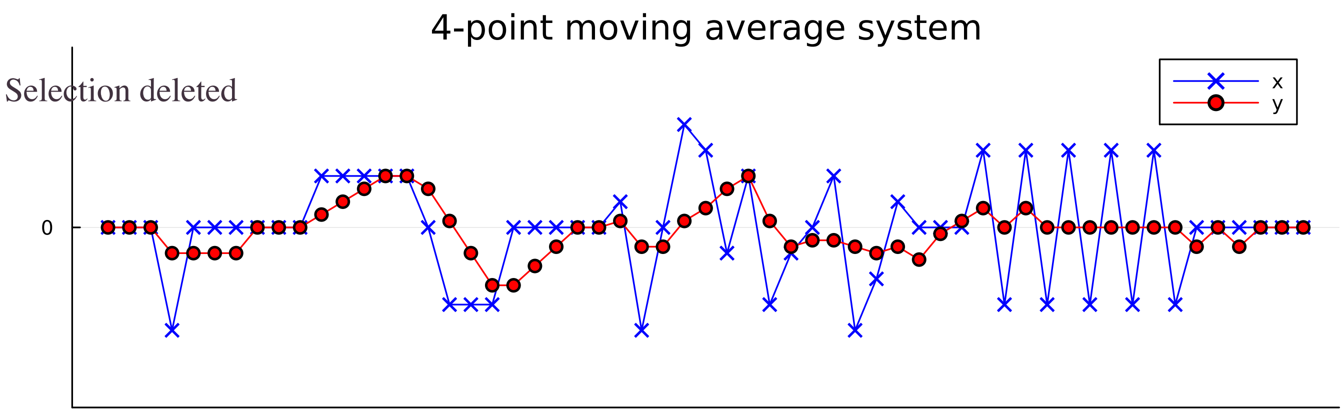
```julia
using DSP, Plots, FFTW
```

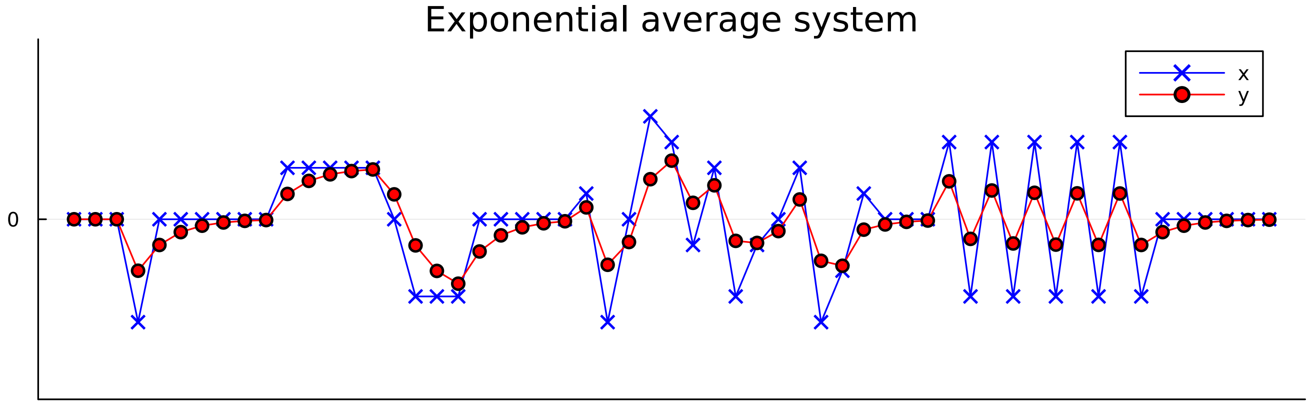
1 (a)

```julia
plot_filter(x, y; kwargs...) =
    plot([x y], linecolor=[:blue :red], markercolor=[:blue :red],
    markershape=[:x :circle], markerstrokewidth=3,
    size = (800, 250), xticks=[], yticks=[0],
    label=["x" "y"], ylim=[-7,7]; kwargs...);
```

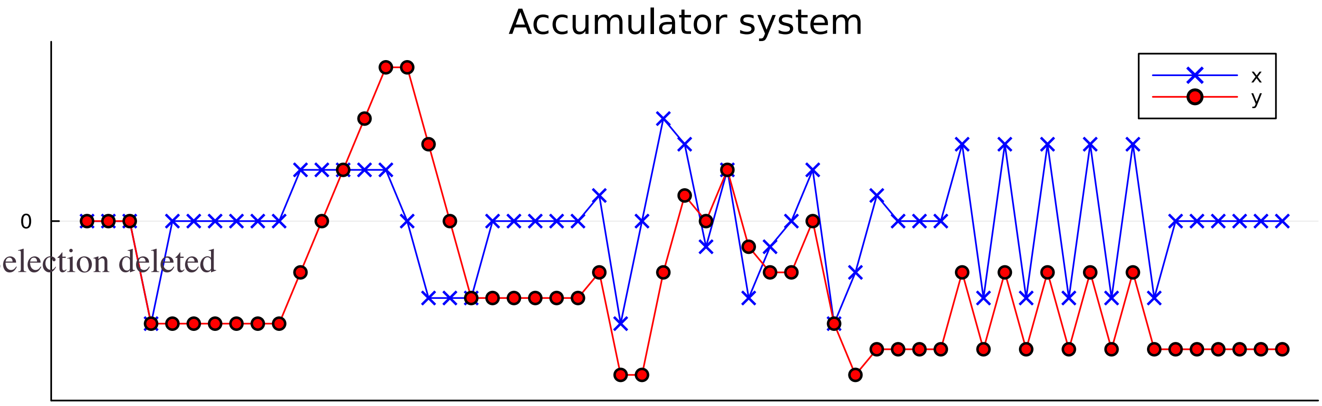
```julia
x = [ 0, 0, 0, -4, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2,
    2, 0, -3, -3, -3, 0, 0, 0, 0, 0, 1, -4, 0, 4,
    3, -1, 2, -3, -1, 0, 2, -4, -2, 1, 0, 0, 0, 3,
    -3, 3, -3, 3, -3, 3, -3, 3, -3, 0, 0, 0, 0, 0, 0 ] .* 1.0;
```




```julia
let
    y = filt([1, 1, 1, 1]/4, [1], x);
    plot_filter(x, y; title="4-point moving average system")
end
```

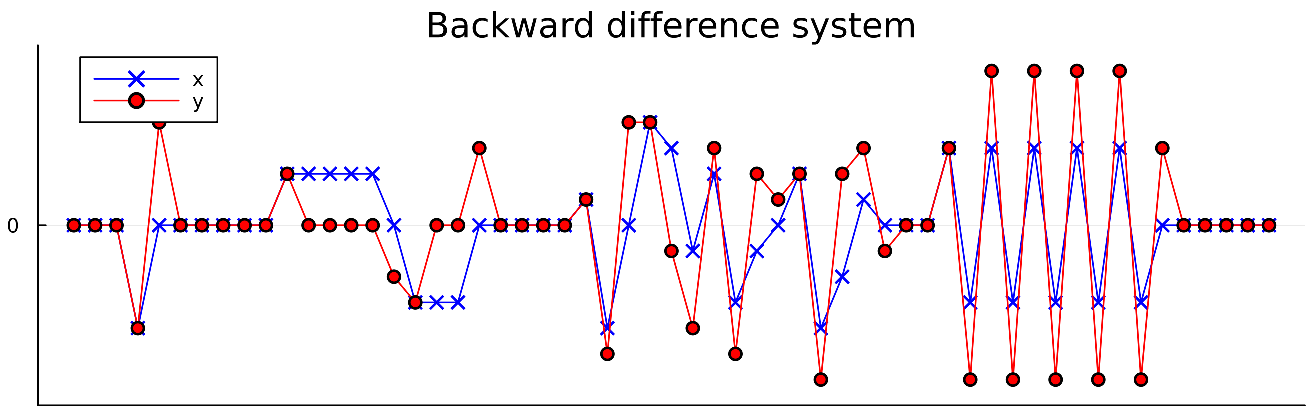



```
let
    α = 0.5
    b = [α]
    a = [1, α-1]
    y = filt(b, a, x);
    plot_filter(x, y; title="Exponential average system")
end
```



Selection deleted

```
let
    b = [1]
    a = [1, -1]
    y = filt(b, a, x);
    plot_filter(x, y; title="Accumulator system")
end
```




```
let
    b = [1, -1]
    a = [1]
    y = filt(b, a, x);
    plot_filter(x, y; title="Backward difference system")
end
```

1 (b)

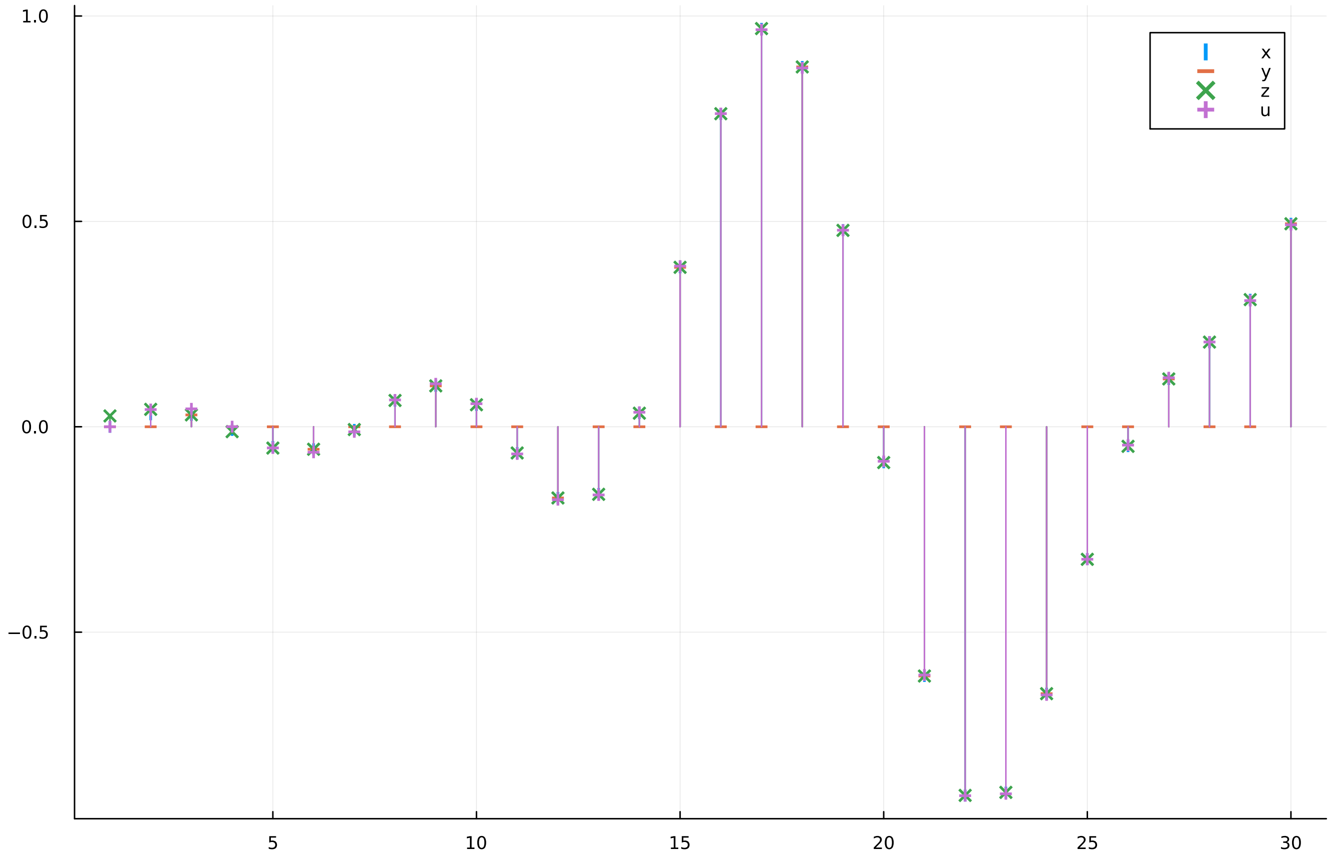
(i)

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.194795, 1.93
```

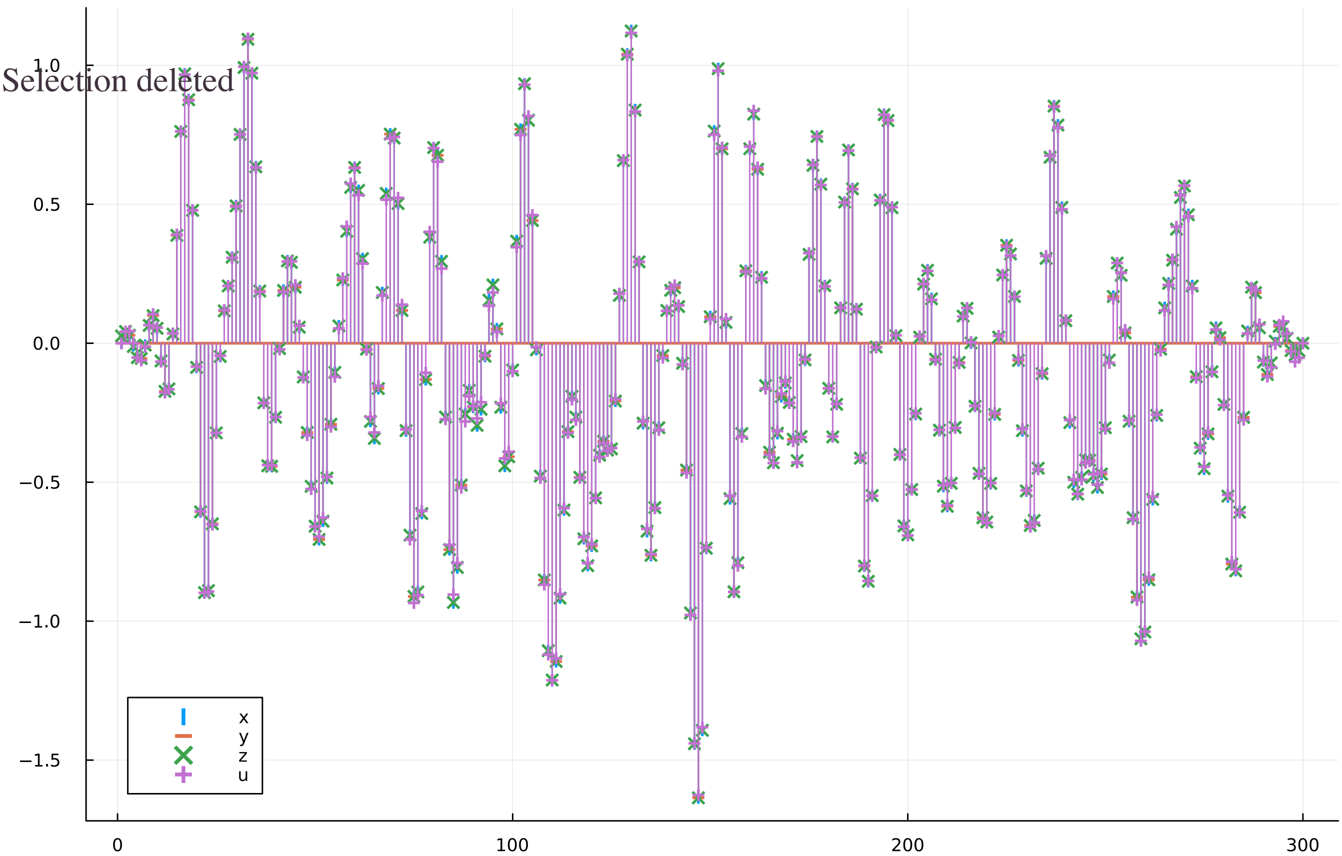
```julia
begin
    # For a 300Hz sampling rate, one second long sequence will have 300 samples
    r = randn(300)
    r[1:15] .= 0
    r[end-14:end] .= 0
    r
end

```

```
sinc_interpolate (generic function with 1 method)
```

Selection deleted



Selection deleted




```julia
let
    # FIR low-pass filter applied on r to obtain x
    filter = digitalfilter(Lowpass(45, fs=300),FIRWindow(Windows.hamming(51)))
    x = filtfilt(filter, r);

    # Sample x at 100Hz to obtain y
    mask = map((x -> mod(x,3) == 0), 1:300)
    y = x .* mask
```

```

    # Sinc interpolation to reconstruct zeroed samples of y
    z = zeros(300)
    for i in (1:300);
        acc = 0.0
            for j in (3:3:length(y));
                acc += (y[j] * sinc((i/3 - j/3)))
            end;
        z[i] = acc;
    end;

    # FIR low-pass filter applied on y to obtain u
    filter_2 = digitalfilter(Lowpass(50, fs=300),FIRWindow(Windows.hamming(51)))

    u = 3 .* filtfilt(filter_2, y)

    p1 = sticks([x[1:30],y[1:30],z[1:30],u[1:30]], size = (900, 600), markershape=
    [:vline :hline :x :+], label=["x" "y" "z" "u"]);
    p2 = sticks([x,y,z,u], size = (900, 600), markershape=[:vline :hline :x :+],
    label=["x" "y" "z" "u"]);
    plot(p1, p2, layout = (2,1), size=(900,1200))
end


```

The vector a required is $[1]$

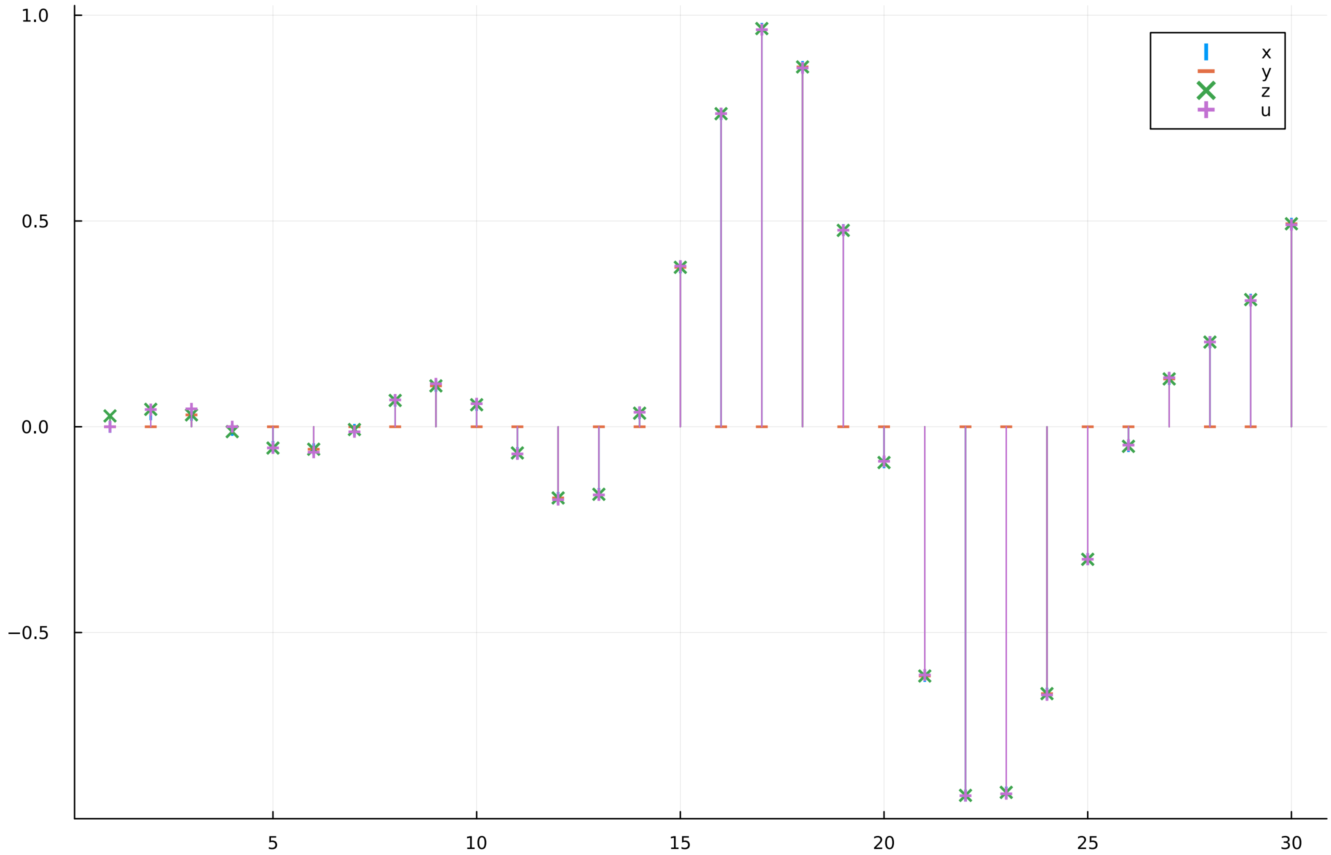
Comparing x and z, they are identical.
Comparing x and u, they are identical.

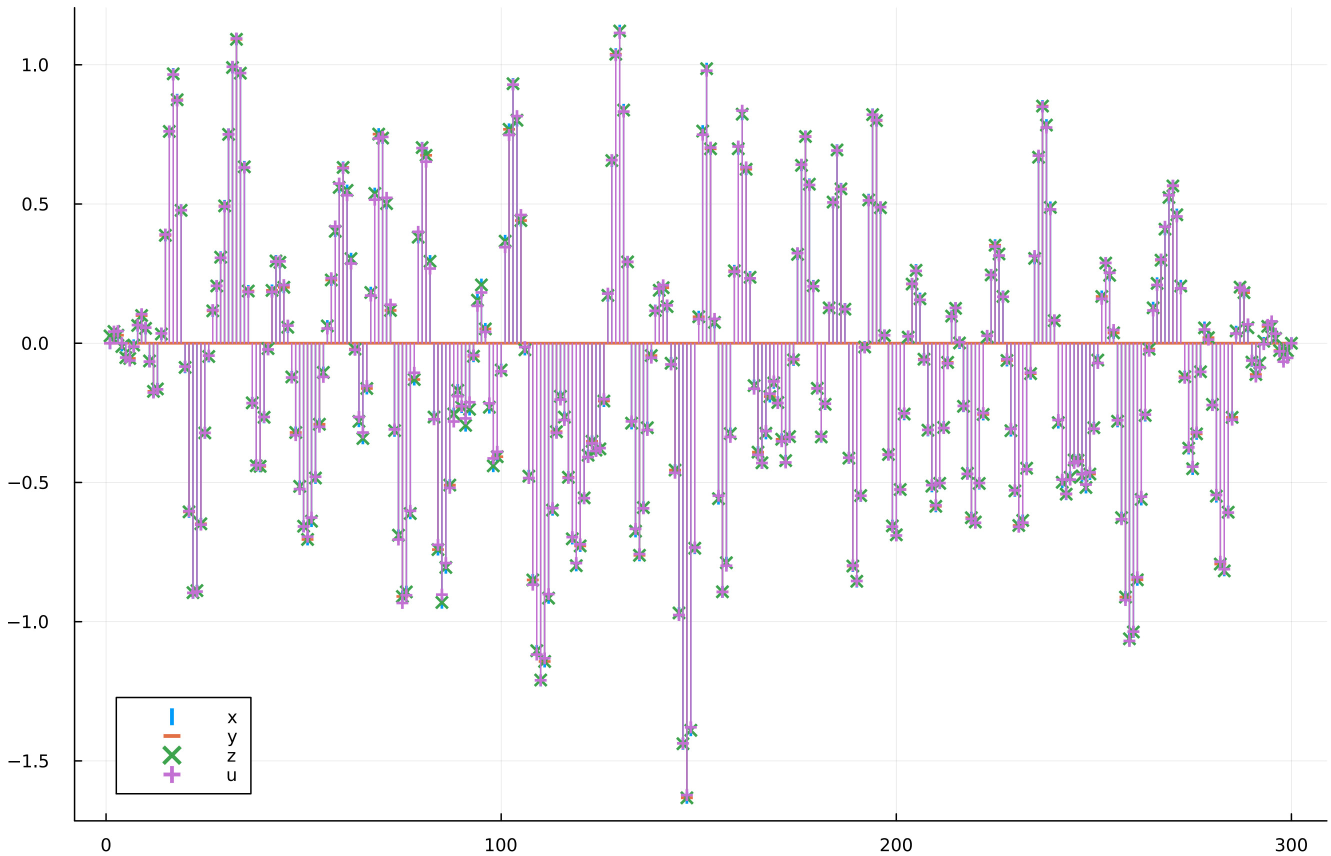
hamming_window (generic function with 1 method)

```
# Create a Hamming window of length n
hamming_window(n) = 0.54 .- 0.46 .* cos.(2 .* π .* (0:n-1)./(n-1))
```

digital_filter (generic function with 1 method)

```
# Create a digital filter
digital_filter(n, fc, fs, w) = 2 .* fc ./ fs .* sinc.(2 .* ((0:n) .- n/2) .* fc ./
fs) .* w
```






```
let
    # FIR low-pass filter applied on r to obtain x
    #filter = digitalfilter(Lowpass(45, fs=300),FIRWindow(Windows.hamming(51)))

    h = hamming_window(51)
    filter = digital_filter(50, 45, 300, h)
    x = filtfilt(filter, r);

```

```
    # Sample x at 100Hz to obtain y
    mask = map((x -> mod(x,3) == 0), 1:300)
    y = x .* mask

    # Sinc interpolation to reconstruct zeroed samples of y
    z = zeros(300)
    for i in (1:300);
        acc = 0.0
            for j in (3:3:length(y));
                acc += (y[j] * sinc((i/3 - j/3)))
            end;
        z[i] = acc;
    end;

    # FIR low-pass filter applied on y to obtain u
    #filter_2 = digitalfilter(Lowpass(50, fs=300),FIRWindow(Windows.hamming(51)))

    filter_2 = digital_filter(50, 50, 300, h)
    u = 3 .* filtfilt(filter_2, y)

    p1 = sticks([x[1:30],y[1:30],z[1:30],u[1:30]], size = (900, 600), markershape=
    [:vline :hline :x :+], label=["x" "y" "z" "u"]);
    p2 = sticks([x,y,z,u], size = (900, 600), markershape=[:vline :hline :x :+],
    label=["x" "y" "z" "u"]);
    plot(p1, p2, layout = (2,1), size=(900,1200))
end
```

(ii)

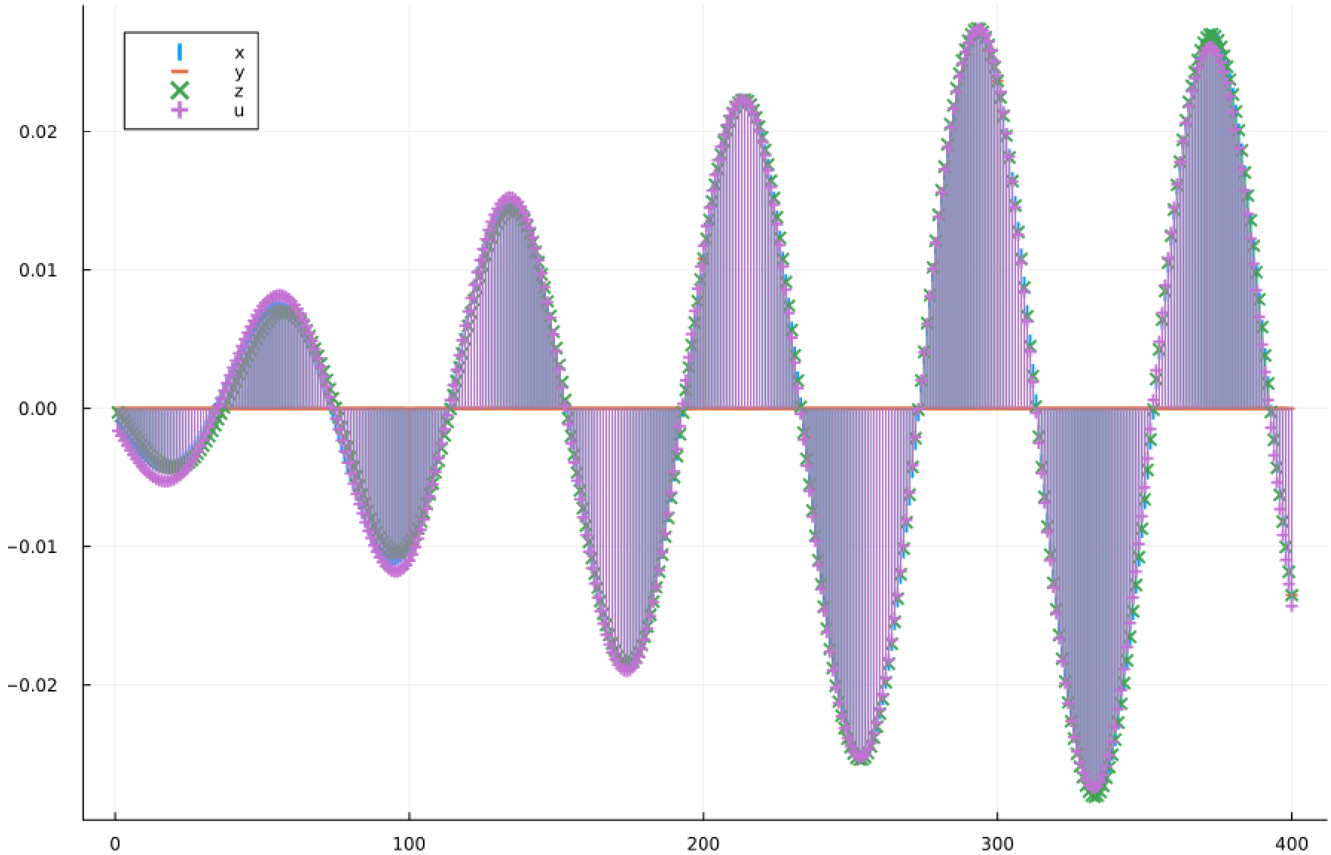
The first filter is an anti-aliasing filter, to restrict the bandwidth of the sequence to the Nyquist limit (i.e., half of the sampling frequency). The sampling theorem tells us that this will result in the aliases in the frequency domain to not overlap, allowing the second filter, a reconstruction filter (at the sampling frequency), to cleanly reconstruct the original signal before sampling. Hence, the first filter should have a lower cut-off frequency than the second.

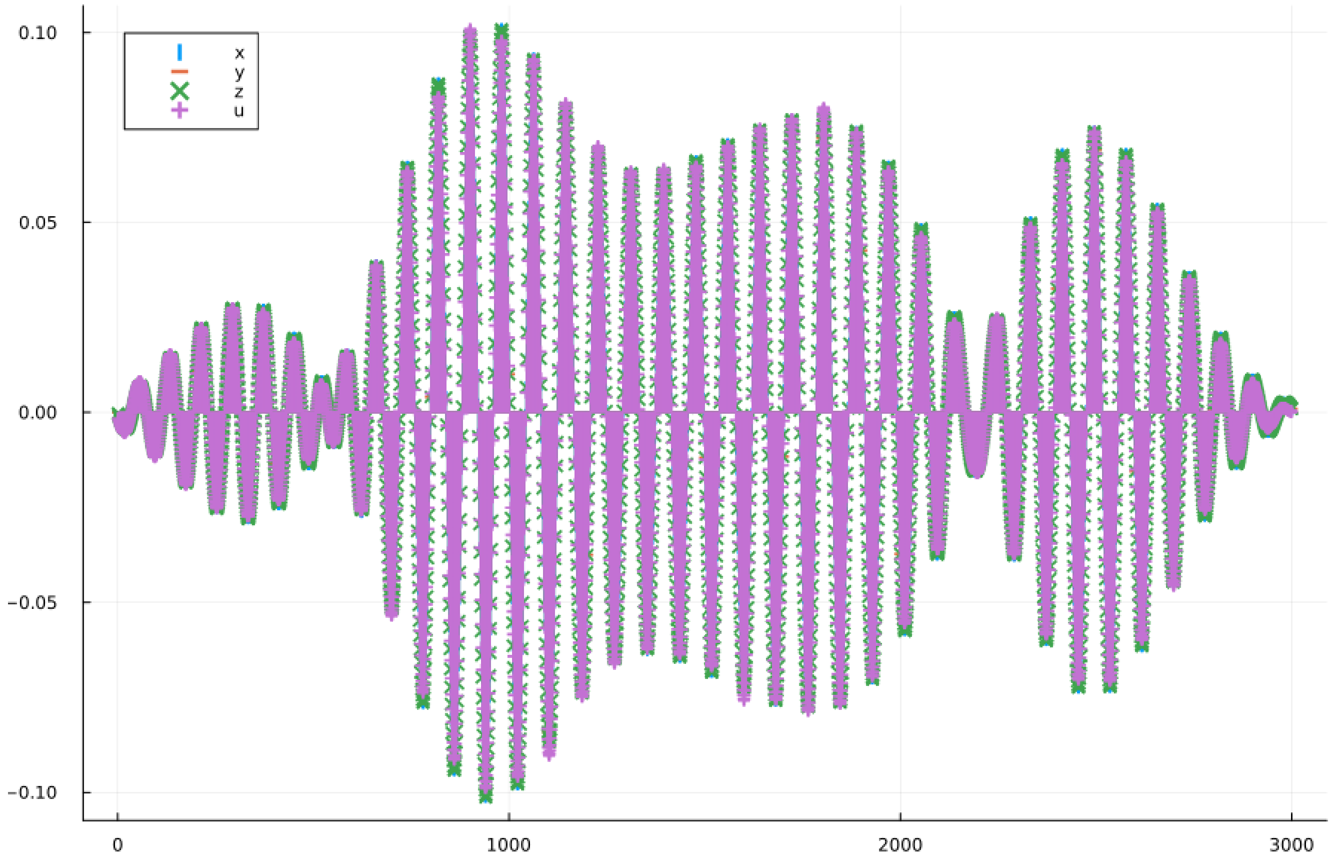
1(c)

(i)

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
```

```julia
begin
    # For a 3kHz sampling rate, one second long sequence will have 3000 samples
    r2 = randn(3000)
    r2[1:500] .= 0
    r2[end-499:end] .= 0
    r2
end
```






```
let
    # Band-pass filter applied on r2 to obtain x
    filter = digitalfilter(Bandpass(31, 44, fs=3000), Chebyshev2(3, 30))
    x = filtfilt(filter, r2);

    # Sample x at 30Hz to obtain y
    mask = map((x -> mod(x,100) == 0), 1:3000)
    y = x .* mask
```

```
    # Sinc interpolation (with modulation) to reconstruct zeroed samples of y
    z = zeros(3000)
    for i in (1:3000);
        acc = 0.0
            for j in (100:100:length(y));
                acc += (y[j] * sinc((i/100 - j/100)/2))* cos(2 * π * ((i-j)*
                1/3000) * 30 * 5 / 4)
            end;
        z[i] = acc;
    end;

    # Band-pass filter applied on y to obtain u
    filter_2 = digitalfilter(Bandpass(30, 45, fs=3000), Chebyshev2(3, 20))
    u = 100 .* filtfilt(filter_2, y)

    p1 = sticks([x[1:400],y[1:400],z[1:400],u[1:400]], size = (900, 600),
    markershape=[:vline :hline :x :+], label=["x" "y" "z" "u"]);
    p2 = sticks([x,y,z,u], size = (900, 600), markershape=[:vline :hline :x :+],
    label=["x" "y" "z" "u"]);
    plot(p1, p2, layout = (2,1), size=(900,1200))
end
```

We have to multiply by a factor of 100 to compensate for the energy lost during sampling $(f_s/f = 3000/30 = 100)$

Comparing x and z, there is some difference at the start, but after some samples, they start to line up.
Comparing x and u, there is some difference at the start, but after some samples, they start to line up.
This is expected because the first 99 samples are set to 0 in order to obtain y from x.

(ii)

If the cut-off frequencies of all the band-pass filters are reduced by $5\ \mathrm{Hz}$, then the spectral components of the sampled signal would not remain in the interval $n \cdot f_s/2 < |f| < (n+1) \cdot f_s/2$ for some $n \in \mathbb{N}$, in particular, it would not remain in the interval $30\ \mathrm{Hz}$ to $45\ \mathrm{Hz}$ (i.e., $n = 2$). This would cause the aliases in the frequency domain after sampling to overlap, and the reconstruction filter (equivalent to sinc interpolation in the time domain) would not be able to cleanly remove the aliases and will not be able to perfectly reconstruct the original signal before sampling.

1 (d)

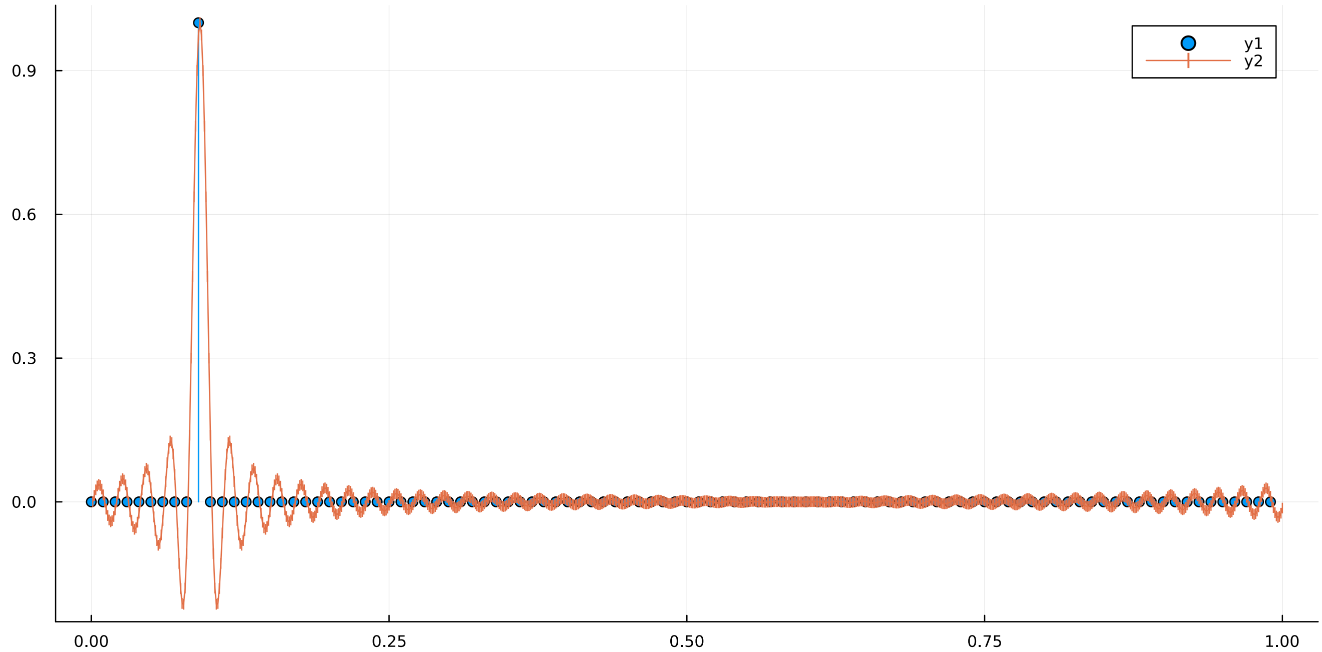
```
fft_interp (generic function with 1 method)
```

```
function fft_interp(x,f)
    if (f <= 0)
        error("fft_interp(x,f): f must be greater than 0")
    end

    pad_length = length(x) * (f-1) -1
    X = (fft(x))
    X_padded = f .* [X[1:50]; 0.5*X[51]; zeros(pad_length); 0.5*X[51]; X[52:100]]

    return real.(ifft((X_padded)))
end

```

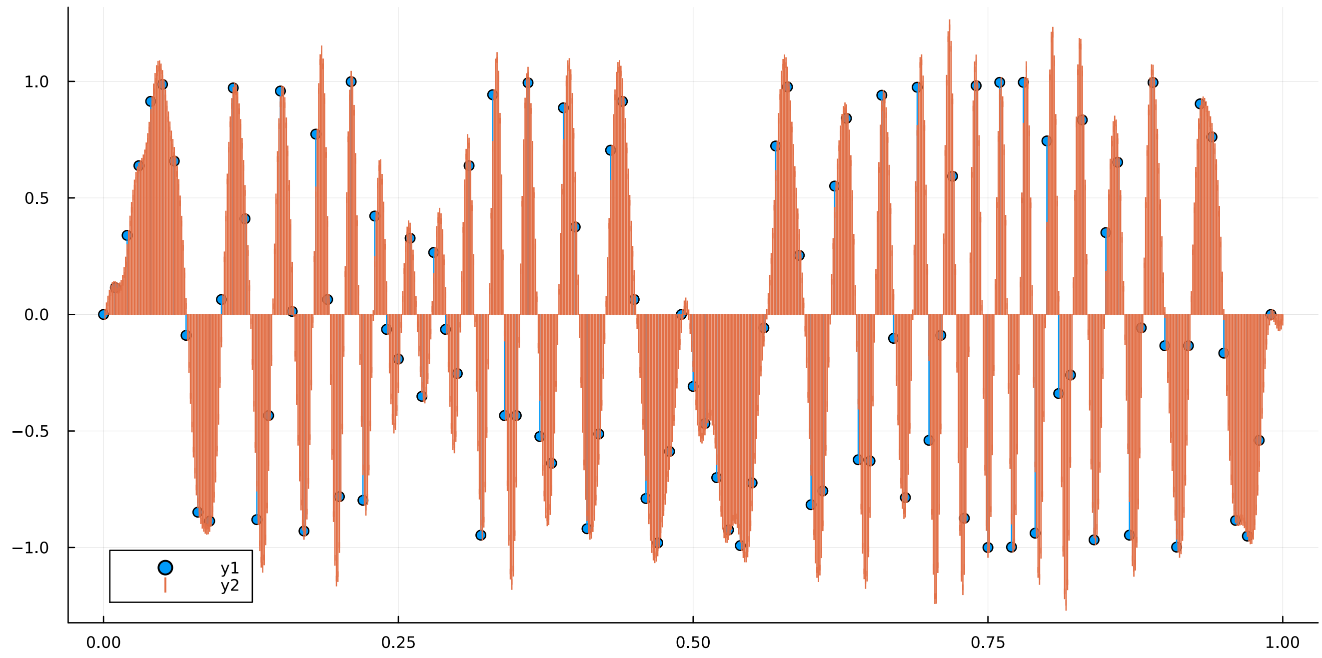
Multiple definitions for fft_interp

Combine all definitions into a single reactive cell using a `begin ... end` block.

```
function fft_interp(x,f)
    if (f <= 0)
        error("fft_interp(x,f): f must be greater than 0")
    end

    time_pad_length = 2 * f * length(x) - length(x) - 1
    x_padded = [x; zeros(time_pad_length)]
    X = (fft(x_padded))
    for i in (1:length(X))
        Printf.@printf("%d %f \n", i, real(X[i]))
    end

    Xx = complex(zeros(length(X)))
    for i in (1:100)
        Xx[i] = X[i]
    end
    for i in (length(X)-100: length(X))
        Xx[i] = X[i]
    end
    Printf.@printf("%d\n", length(Xx))

    return (real.(ifft((Xx))))[1:f*length(x)]
end
```




```
let
    f = 8
    x = (0:99)/100
    y = zeros(100)
    y[10] = 1

    x_i = (1:f*100)/(f*100)
    y_i = fft_interp(y,f)
    sticks(x, y, shape=:circle, size=(1000,500))
    plot!(x_i, y_i, shape=:vline)
end
```




```
let
    x = (0:99)/100
    rad_per_sample = [0.9 ./ 49 .* π .* (0:49) ; 0.9 ./ 49 .* π .* (49:-1:0)]
    y = sin.( rad_per_sample .* (1:100))

    f = 8

    x_i = (1: f*100)/(f * 100)
    y_i = fft_interp(y,f)
    sticks(x, y, shape=:circle, size=(1000,500))
    sticks!(x_i, y_i, shape=:vline)
end

```

I realised I might have approached this entirely wrongly. I was thinking about the slides which talked about zero padding to increase resolution/sampling of DTFT to obtain DFT, and hence thought of using zero padding in frequency domain to obtain higher sampling rate in the time domain, but this doesn't solve the issue of wrapover.

Instead, I should have done padding in the time domain, and then multiply with a rect in the freq domain (equivalent to convolving with a sinc in the time domain), and then converting back to the time domain & remove the padding. I realised this a bit too late though, so I just have to submit what I have done.