5650 Fall 2022_Set 1_Julia_Demo

August 29, 2022

Contents

[1]:
```
import Plots, PyCall, DSP
```

[2]:
```
ss = PyCall.pyimport("sk_dsp_comm.sigsys")
```

[2]:
```
PyObject <module 'sk_dsp_comm.sigsys' from
'/Users/mwickert/.julia/conda/3/lib/python3.8/site-
packages/sk_dsp_comm/sigsys.py'>
```

[3]:
```
"""
    ingredients(path::String)

This function is used to import a user code module into the current workspace.
"""
function ingredients(path::String)
    # this is from the Julia source code (evalfile in base/loading.jl)
    # but with the modification that it returns the module instead of the last␣
 ↪object
    name = Symbol(basename(path))
    m = Module(name)
    Core.eval(m,
        Expr(:toplevel,
             :(eval(x) = $(Expr(:core, :eval))($name, x)),
             :(include(x) = $(Expr(:top, :include))($name, x)),
             :(include(mapexpr::Function, x) = $(Expr(:top, :include))(mapexpr,␣
 ↪$name, x)),
```

```
            :(include($path))))
    m
end
```

[3]: ingredients

[4]:
```
DSPTools = ingredients("modules/DSPTools.jl")
```

[4]: Main.DSPTools.jl

Section 1 of a New Notebook

Consider a simple model for the signal

$$x[n] = \delta[n-2]$$

or by another means

$$x[n] = u[n-2] - u[n-3] \overset{\text{also}}{=} \delta[n-2]$$

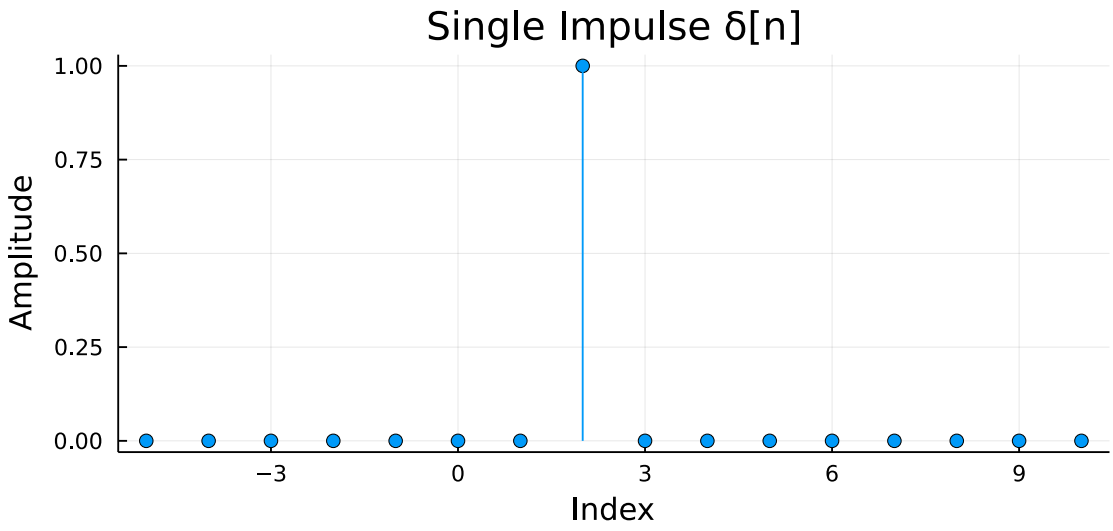
Too create a very simple Python or Julia model we can use signal primitives found in the module `scikit-dsp-comm.sigsys`. In an earlier code cell the module was loaded into the Python workspace using

```
import sk_dsp_comm.sigsys as ss
```

An *alias* of `ss` is created to be more convenient for typing.

[10]:
```
# A code cell with code
n = collect(-5:10)
# x = ss.dstep(n-2) - ss.dstep(n-3)
x = ss.dimpulse(n .- 2)
DSPTools.stem(n,x; title="Single Impulse  [n]")
# Plots.plot!(ylim([-.1,1.1]))
```

[10]:



Worked Problem

Text 2.3

Consider

$$y[n] = x[n] * h[n]$$

where

$$x[n] = u[n] \tag{1}$$
$$h[n] = a^{-n}u[-n], \text{ with } a = 0.8 \tag{2}$$
$$\tag{3}$$

Again we create the two signals with the help of `sigsys.py` (aliased as `ss`) and move on to the use of a helper function `ss.conv_sum()` for computing the convolution of two sequences. The scipy module `scipy.signal` alised as `signal` contains the function `convolve` but `ss.conv_sum` incorporates a means for properly tracking sequence axis n for proper plotting of output.

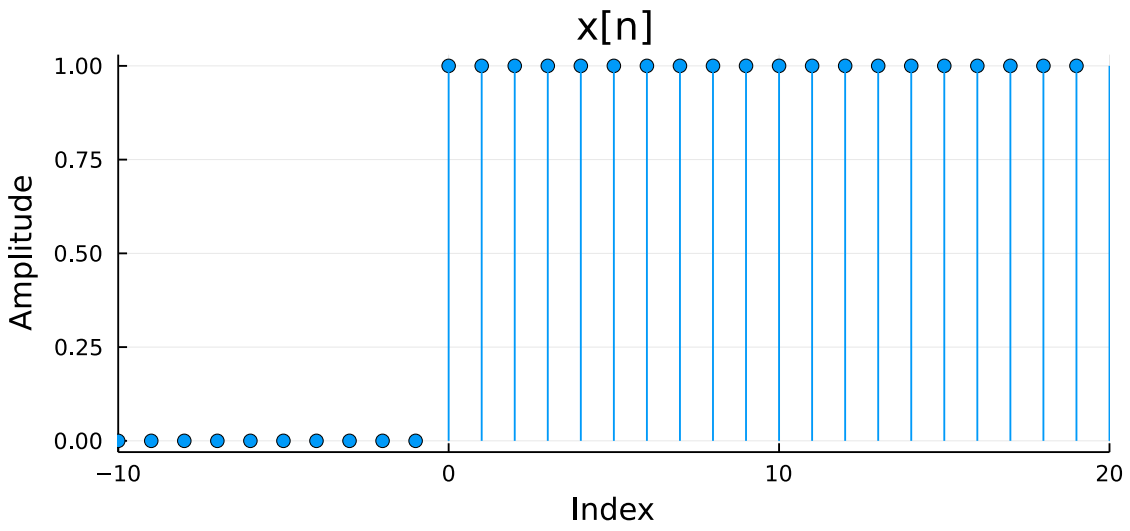
Create and Plot the Signals $x[n]$ and $h[n]$

Note each of these signals has semi-infinite duration: $x[n]$ is right-sided and $h[n]$ is left-sided.

[13]:
```
a = 0.8
nx = collect(-10:40)
x = ss.dstep(nx)
nh = collect(-20:10)
h = a.^(-nh) .* ss.dstep(-nh);
```

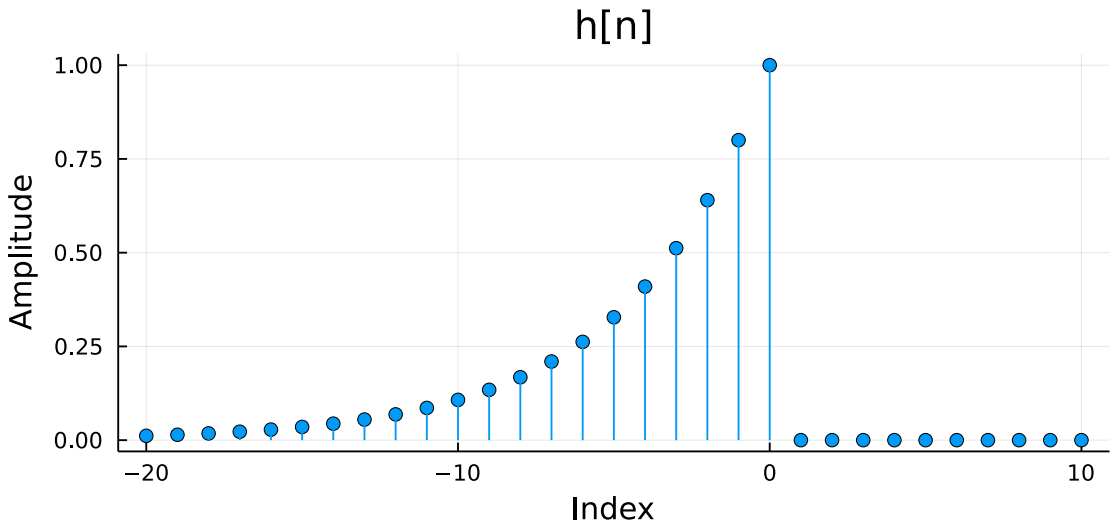
[34]:
```
DSPTools.stem(nx,x;title="x[n]")
Plots.plot!(xlim=(-10,20))
```

[34]:



[35]:
```
DSPTools.stem(nh,h;title="h[n]")
# Plots.plot!(ylim=(-0.5,1.05))
```

[35]:



Form $y[n]$ Using the Convolution Sum

In code the convolution sum is a numerical calculation which typically is run over a finite range of index values k and output values n. Note for each output index n you must sum over a given range of input values $k \in [N_1, N_2]$. A consequence of forming the convolution sum over a finite set of index values, when one or both sequences **do not have finite duration**, is there may be end effects in the resulting output $y[n]$. The function `ss.conv_sum` resolves by supressing output values at one or both ends with the use of the optional `extent` arguments: `r` for right-sided, 'l' for left-sided, and 'f' (default) for finite length sequences.

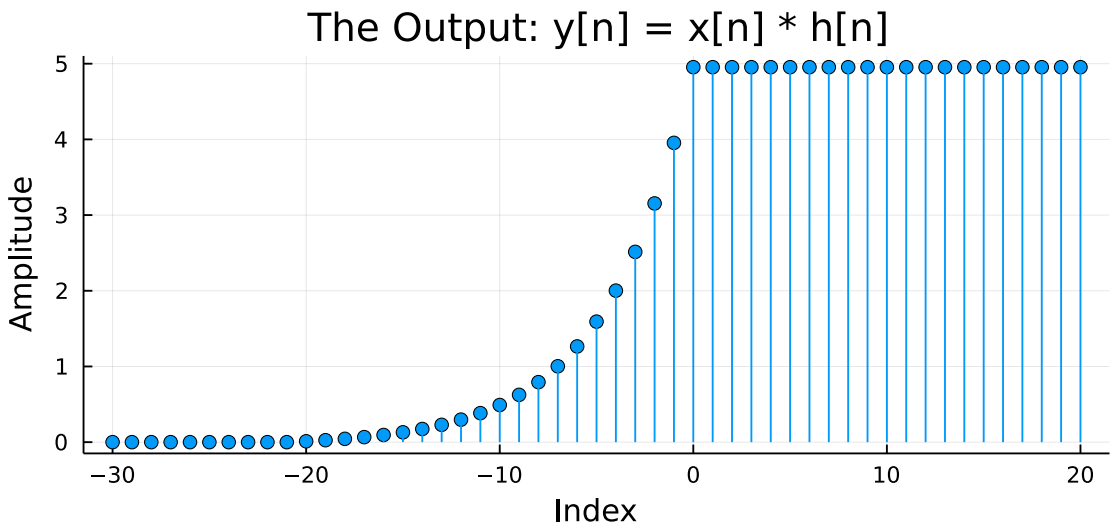
In the below we use `r` for $x[n]$ and `f` for $h[n]$ and achieve the desired results. You show this theoretically in working problem 2.3.

[18]:
```
y, ny = ss.conv_sum(x,nx,h,nh,extent=("r","f"));
```

```
Output support: (-30, +20)
```

[19]:
```
DSPTools.stem(ny,y;title="The Output: y[n] = x[n] * h[n]")
```

[19]:


The Output: y[n] = x[n] * h[n]
Amplitude
Index
0
1
2
3
4
5
−30
−20
−10
0
10
20

Text 2.30

In this problem you consider a system that is a cascade of the two LTI systems:

$$h_1[n] = u[n] - u[n-4] \tag{4}$$
$$h_2[n] = u[n+3] - u[n-1] \tag{5}$$
$$\tag{6}$$

This time the simulation will be done using the `scipy.signal` difference equation engine `y = lfilter(b,a,x)`. In Julia you use the function `DSPTools.lccde(b,a,x)`. Since $h_2[n]$ is non-causal (why?) we will have to adjust the time axis to work the problem assuming causal filters, but then shifting the axis to the left by 3 when considering outputs $w[n]$ and $y[n]$. So, in modeling the two filter impulse responses we start by shifting $h_2[n]$ to the right by 3 samples:

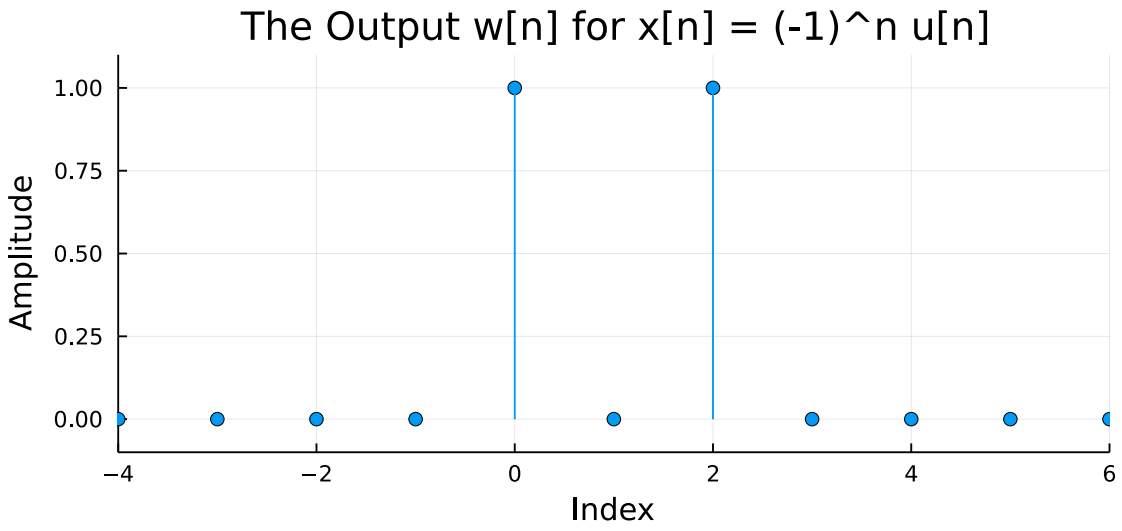
[23]:
```
# Causal impulse responses start at n = 0
h1 = [1, 1, 1, 1]
h2 = [1,1,1,1];
```

Part a

Here we let $x[n] = (-1)^n u[n]$

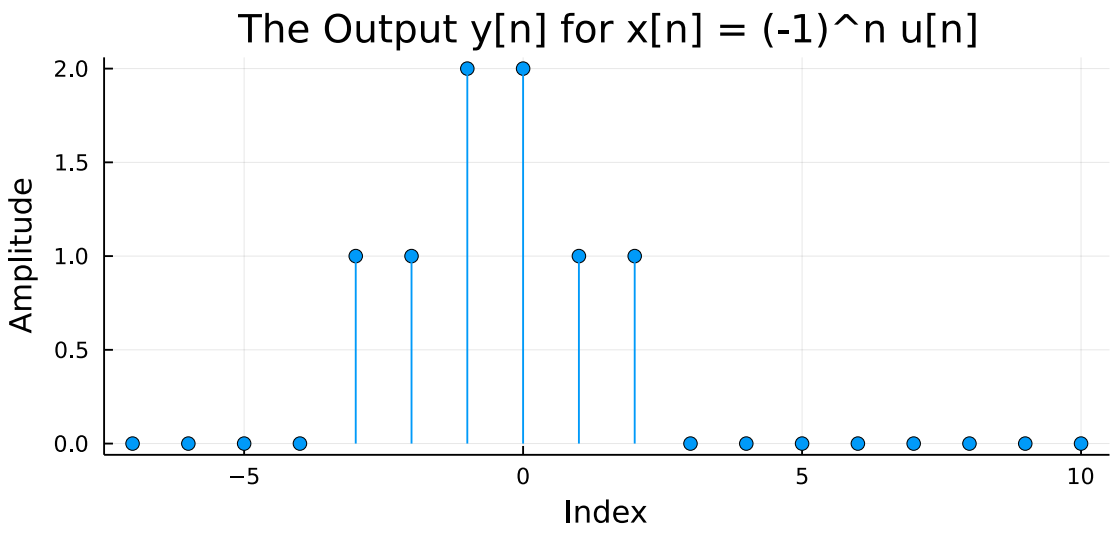
[29]:
```
# The input does not shift, just the system impulse responses
n = collect(-4:13)
xa = (-1.0).^n .* ss.dstep(n)
w = DSPTools.lccde(h1,[1],xa)
y = DSPTools.lccde(h2,[1],w)
# Plot the outputs but shift the time axis back to the left by 3 after h2

DSPTools.stem(n,w;title="The Output w[n] for x[n] = (-1)^n u[n]")
Plots.plot!(xlim=(-4,6),ylim=(-0.1,1.1))
```

[29]:



[28]:
```
DSPTools.stem(n .- 3,y,title="The Output y[n] for x[n] = (-1)^n u[n]")
```

[28]:



Part b

Carry on with $x[n] = \delta[n]$ and find the overall impulse response.

Part c & d

Carry on with $x[n] = 2\delta[n] + 4\delta[n-4] - 2\delta[n-12]$ and plot $w[n]$ and $y[n]$.

[ ]: