To: ACS Handbook
From: Eric Stoneking
Date: Dec 2021
Subject: Example of Frequency Response using Julia in a Jupyter Notebook

Introduction

This notebook demonstrates a frequency response analysis using only fundamental operations on complex numbers. Our system is simple, but features both continuous-time and discrete-time elements. The purpose of this notebook is as much about familiarization with julia as it is about the analysis.

Equations

The rigid-body transfer function is

$$\text{Rigid} = \frac{1}{Is^2}. \tag{1}$$

We use a PID control law, $K_r s + K_p + K_i/s$. We add a flex mode transfer function,

$$\text{Flex} = \frac{\phi}{s^2 + 2\zeta_f \omega_f s + \omega_f^2} \tag{2}$$

Discrete-time elements use $z = e^{sT}$. A zero-order hold is modeled as

$$\text{ZOH} = \frac{1 - 1/z}{sT} \tag{3}$$

and a one-cycle delay is $1/z$.

Finding the Frequency Response

We'll do the frequency response twice, to showcase julia vectorizing syntax. The first time, we set up a for-loop over frequency. The second time, we do it with vectorization and broadcasting functions.

The second cell overwrites the results of the first. In this notebook, though, you can selectively run the first one and skip the second if you would like to check the results in subsequent plots. The focus of this demonstration is more about comparing the syntaxes.

```julia
# System Parameters
I = 100.0;
wc = 0.1*2*pi;
zc = 0.7;
Kr = 2.0*I*zc*wc;
Kp = I*wc^2;
Ki = 0.01*Kp;
T = 0.25; # Sample time
PF = 0.001; # Flex mode participation factor
wf = 2.0*2.0*pi;
zf = 0.005;
```

```
### Frequency Response
# C-style
Nf = 3001;
f = zeros(Nf,1);
dB = zeros(Nf,1);
Phase = zeros(Nf,1);
for i=1:Nf
    f[i] = 10.0^(-2+0.001*(i-1));
    w = 2*pi*f[i];
    s = im*w;  # "im" is julia's imaginary unit
    z = exp(s*T);
    ZOH = (1-1/z)/(s*T);
    Delay = 1/z;
    PID = Kr*s+Kp+Ki/s;
    RigidBody = 1/(I*s^2);
    FlexBody = PF/(s^2+2*zf*wf*s+wf^2);
    TF = ZOH*Delay*PID*(RigidBody+FlexBody);
    # TF is a complex number.  Find its magnitude and phase.
    Mag = abs(TF);
    dB[i] = 20.0*log10(Mag);
    Phase[i] = angle(TF)*180.0/pi;
end
```

```
# Same thing as above, vectorized julia-style.
# Note use of dots to broadcast functions, even + and -
f = 10.0.^Vector(-2:0.001:1);
Nf = length(f);
w = 2*pi*f;
s = im*w;
z = exp.(s*T);
ZOH = (1.0 .- 1.0./z)./(s*T);
Delay = 1.0./z;
PID = Kr*s .+ Kp .+ Ki./s;
RigidBody = 1.0./(I*s.^2);
FlexBody = PF./(s.^2 .+ 2.0*zf*wf*s .+ wf.^2);
TF = ZOH.*Delay.*PID.*(RigidBody+FlexBody);
Mag = abs.(TF);
dB = 20.0*log10.(Mag);
Phase = angle.(TF)*180.0/pi;
```

```julia
# Unwrap vector x with period p
function unwrap(x,p)
   if length(x) > 1
      for i=1:length(x)-1
         if (x[i+1] > x[i]+0.5*p)
            x[i+1] -= p;
         elseif (x[i+1] < x[i]-0.5*p)
            x[i+1] += p;
         end
      end
   end
   return(x)
end;
```

```julia
# Massage Phase for clear viewing
if (Phase[1] > 0.0)
   Phase = Phase .- 360.0;
end
Phase = unwrap(Phase,360.0);
```

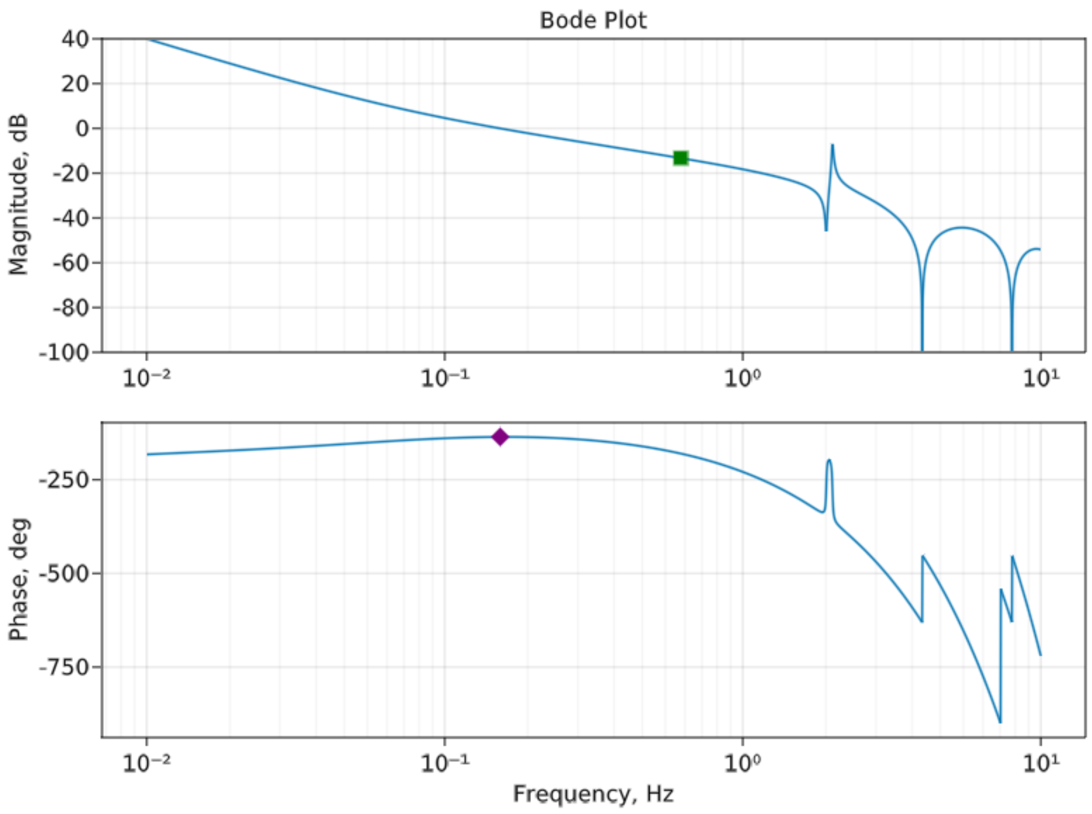
```julia
# Find crossover points
i=1
while dB[i] > 0.0
   i += 1;
end
f_pm = f[i];
Phase_pm = Phase[i];
TF_pm = TF[i];

while Phase[i] > -180.0
   i += 1;
end
f_gm = f[i];
Mag_gm = Mag[i];
dB_gm = dB[i];
TF_gm = TF[i];
```

Plots of Results

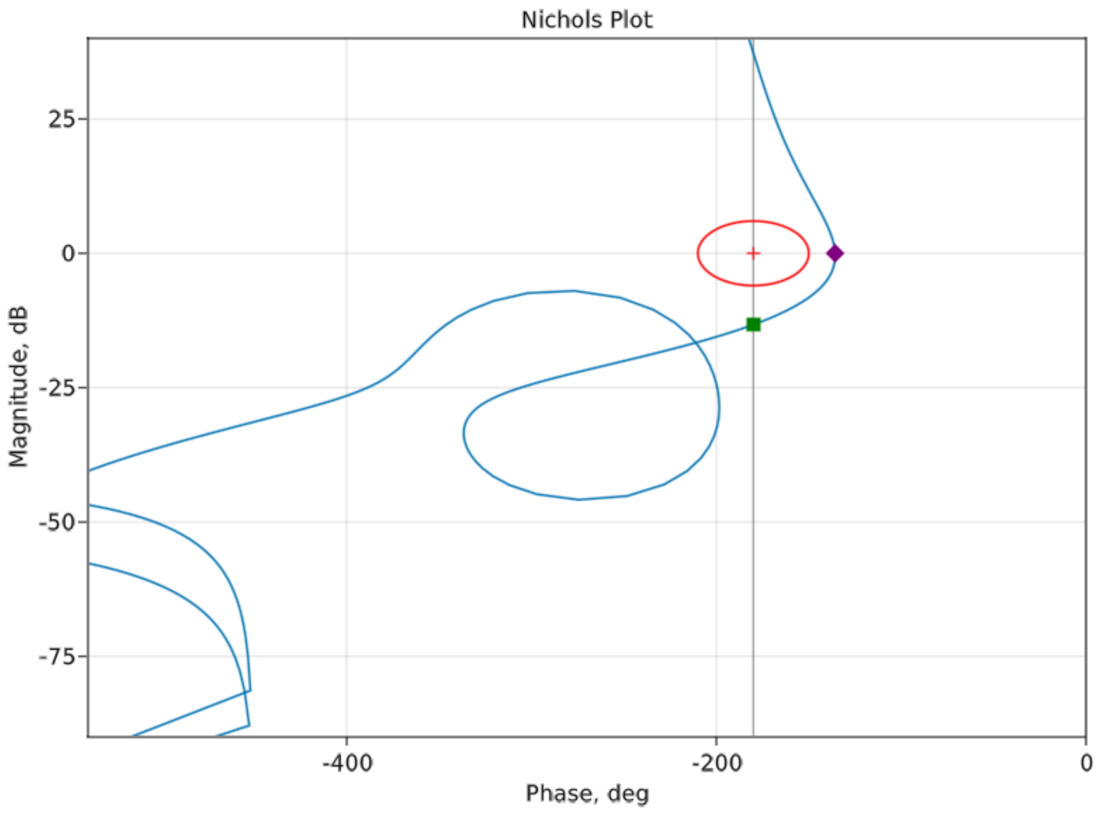
```julia
using GLMakie # Plotting package.  Packages add in a lot of functionality.
```

```
BodePlot = Figure();
   MagAxis = Axis(BodePlot[1,1],xscale=log10);
      ylims!(MagAxis,-100,40);
      MagAxis.xminorgridvisible = true;
      MagAxis.xminorticks = IntervalsBetween(10, true)
      MagAxis.title="Bode Plot";
      MagAxis.ylabel="Magnitude, dB";
      lines!(MagAxis,f,dB);
      scatter!(MagAxis,[f_gm],[dB_gm],color=:green,marker=:rect,markersize=10)
      # For a list of axis foo's attributes, enter "foo.attributes"
   PhaseAxis = Axis(BodePlot[2,1],xscale=log10);
      PhaseAxis.xlabel="Frequency, Hz";
      PhaseAxis.ylabel="Phase, deg";
      PhaseAxis.xminorgridvisible = true;
      PhaseAxis.xminorticks = IntervalsBetween(10, true)
      lines!(PhaseAxis,f,Phase);
      scatter!(PhaseAxis,[f_pm],[Phase_pm],color=:purple,marker=:
 ↪diamond,markersize=14)
BodePlot
```



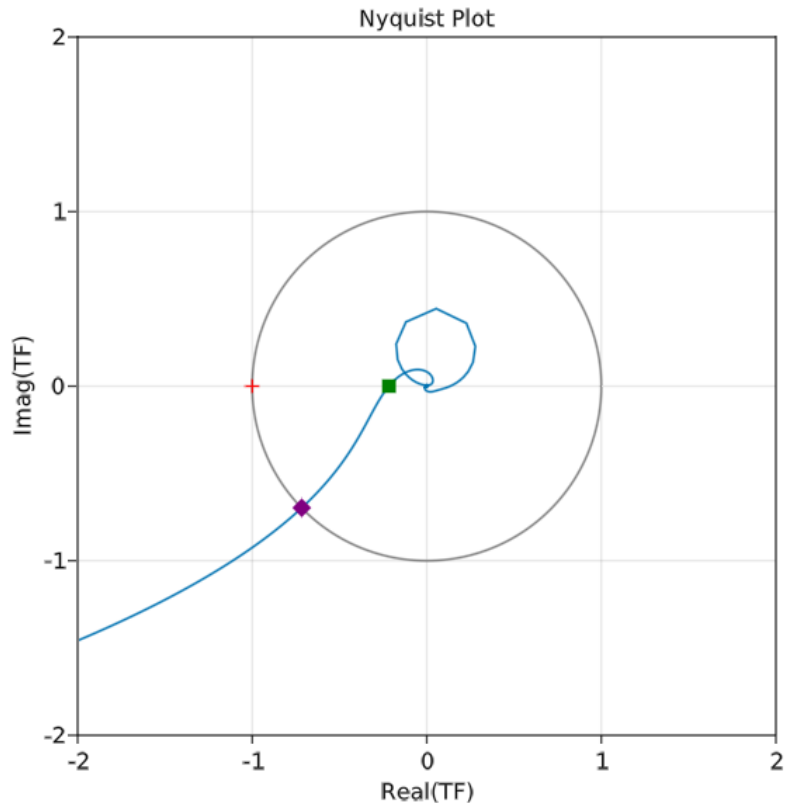
The Bode plot shows magnitude and phase vs. frequency. The gain margin may be read from where phase crosses -180 deg (the green square), and the phase margin may be read from where gain crosses 0 dB (the purple diamond).

```
NicholsPlot = Figure();
   ax = Axis(NicholsPlot[1,1]);
   xlims!(-540,0);
   ylims!(-90,40);
   ax.title="Nichols Plot";
   ax.xlabel="Phase, deg";
   ax.ylabel="Magnitude, dB";
   lines!([-180;-180],[40;-90],color=:gray,linewidth=1)
   scatter!([-180],[0],marker='+',color=:red)
   a = Vector(0:pi/100:2*pi);
   a = [a;a[1]];
   lines!(-180 .+30*cos.(a),6*sin.(a),color=:red)
   lines!(Phase,dB);
   scatter!([-180],[dB_gm],color=:green,marker=:rect,markersize=10)
   scatter!([Phase_pm],[0],color=:purple,marker=:diamond,markersize=14)
NicholsPlot
```



The Nichols plot is magnitude vs. phase. The gain and phase margins are the same as in the Bode plot, but easier to read because they're on the same plot.

```
NyquistPlot = Figure();
   ax = Axis(NyquistPlot[1,1],aspect=1);
   xlims!(-2,2);
   ylims!(-2,2);
   ax.title="Nyquist Plot";
   ax.xlabel="Real(TF)";
   ax.ylabel="Imag(TF)";
   a = Vector(0:pi/100:2*pi);
   a = [a;a[1]];
   lines!(cos.(a),sin.(a),color=:gray);
   lines!(real(TF),imag(TF));
   scatter!([-1],[0],marker='+',color=:red,markersize=10);
   scatter!([real(TF_gm)],[imag(TF_gm)],color=:green,marker=:rect,markersize=10)
   scatter!([real(TF_pm)],[imag(TF_pm)],color=:purple,marker=:
 ↪diamond,markersize=14)
NyquistPlot
```



The Nyquist plot is just the transfer function itself plotted in the complex plane. The gain and phase margins may be deduced from where the transfer function crosses the unit circle, and where it crosses the negative real axis. I don’t use the Nyquist chart much, but I think it gives the clearest intuition when we generalize to MIMO stability analysis.