Python 6 assignment
Laplace and Fourier Transforms

November 4, 2013

In this assignment, we will look at how to analyse “Linear Time-invariant Systems” with numerical tools in Python. LTI systems are what Electrical Engineers spend most of their time thinking about - linear circuit analysis or communication channels for example.

1 The Assignment

1. Work through the example code and understand how to simulate LTI systems.
2. Solve for the time response of a spring satisfying

$$\ddot{x} + x = 0$$

with $x(0) = 0.1$ and $\dot{x} = 0$ for $t$ going from zero to 20 seconds. **Hint**: Remember that initial conditions are part of the Laplace transform. To get the response you can either use the trick I did, or use the impulse method (i.e., if your system is `sci1` then use `sci1.impulse`). Note that since the initial conditions will be part of the transfer function, `X0` will be zero and can be omitted. Compare with theory.

First we set up python.

1a $\langle$*Q2* 1a$\rangle\equiv$

```
from scipy import *
from matplotlib.pyplot import *
import scipy.signal as signal
# time constant is unity. So step of 0.1 is ok.
t=linspace(0,20,201)
```

The Laplace transform of the equation yields

$$s^2X - sx(0) - \dot{x}(0) + X = 0$$

Hence,

$$X(s) = \frac{sx(0) + \dot{x}(0)}{s^2 + 1} = \frac{s^2x(0) + s\dot{x}(0)}{s^2 + 1} \times \frac{1}{s}$$

1b $\langle$*Q2a* 1b$\rangle\equiv$

```
num=poly1d([0.1,0,0])
den=poly1d([1,0,1])
sys1=signal.lti(num,den)
y=sys1.step(T=t)[1]
figure(0)
plot(t,y,’r’)
xlabel(r’$t$’)
title("Q2: Time Response using sys1.step")
```

2 $\langle$*Q2b* 2$\rangle\equiv$

```
num=poly1d([0.1,0])
den=poly1d([1,0,1])
sys1=signal.lti(num,den)
y=sys1.impulse(T=t)[1]
figure(1)
plot(t,y,'r')
xlabel(r'$t$')
title("Q2: Time Response using sys1.impulse")
```

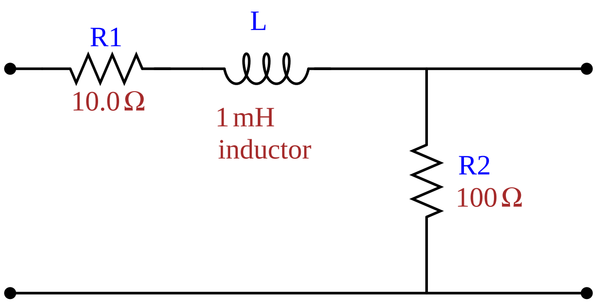
Either of the above blocks does the trick. One uses the step response and the other uses the impulse response.

3. Solve for a coupled spring problem:

$$\begin{aligned}\ddot{x}+(x-y) &= 0\\ \ddot{y}+2(y-x) &= 0\end{aligned}$$

where the initial condition is $x(0)=1$, $\dot{x}(0)=y(0)=\dot{y}(0)=0$. Substitute for y from the first equation into the second and get a fourth order equation. Solve for its time evolution, and from it obtain $x(t)$ and $y(t)$ for $0\le t\le 20$.

4. Obtain the magnitude and phase response of the Transfer function of the following two-port network.

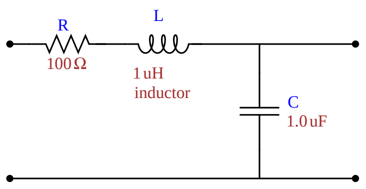


5. Consider the problem in Q4. suppose the input signal $v_i(t)$ is given by

$$v_i(t)=\begin{cases}\cos\omega_0 t & |t|<\pi/2\omega_0\\ 0 & \text{else}\end{cases}$$

where ω_0 is to be chosen to lie where the transform is changing fastest. Obtain the output voltage $v_0(t)$ using numerical Fourier Transforms. Sample the input waveform at 512 time points between 0 and 50μs and do your calculations.

2 Network Analysis



We start with a 2-port network. The transfer function is

$$\begin{aligned}V_0 &= \frac{1/sC}{sL+R+1/sC}V_i\\ &= \frac{1}{s^2LC+sCR+1}V_i\end{aligned}$$

Here the capacitor is assumed ideal, but the inductor has coil resistance which is the R. What is the frequency response of this network? What is the time response to a step input? This is what Fourier and Laplace analysis is good for.

Frequency Response

We convert from s-domain to ω-domain by writing $s = j\omega$. Our transfer function becomes

$$V_0 = \frac{1}{(1-\omega^2 LC) + j\omega RC} V_i$$

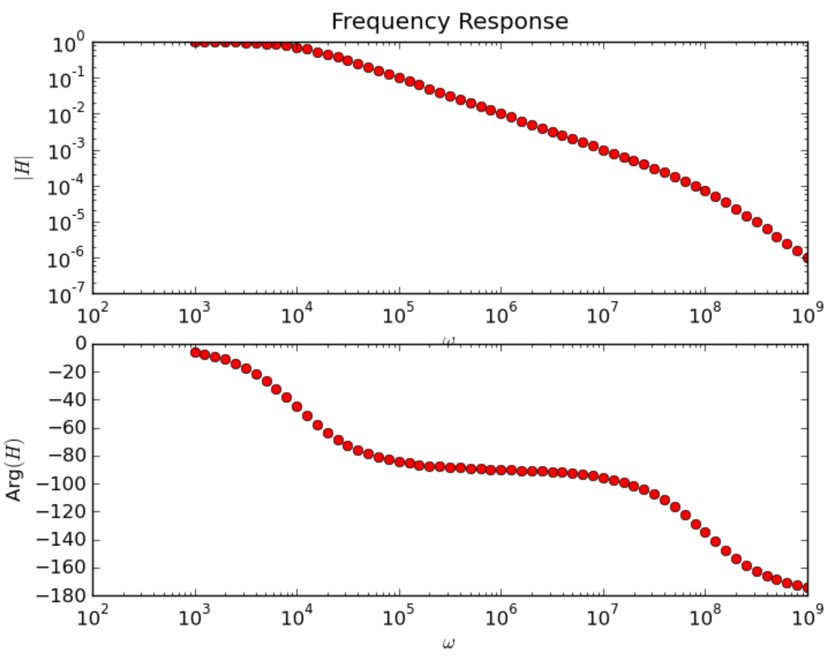
Hence,

$$\frac{V_0}{V_i} = \frac{1}{(1-\omega^2 LC) + j\omega RC}$$

Let us plot this frequency response.

4 $\langle$*eg1* 4$\rangle\equiv$ 5▷

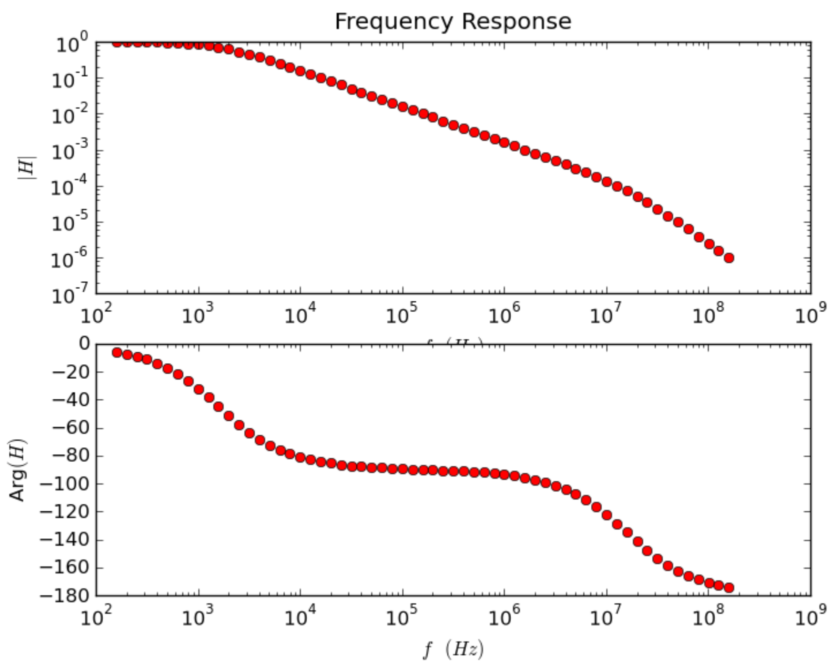
```
from scipy import *
from matplotlib.pyplot import *
L=1e-6   # L is 1\mu
H
C=1e-6   # C is 1\mu
F
R=100    # R is 100\Omega

omega=logspace(3,9,61).reshape((61,1))  # 4
 decades in 41
 steps
H=1/((1-omega*omega*L*C) + 1j*omega*R*C)
figure(1)
cla()
subplot(211)
loglog(omega,abs(H),'ro')
title("Frequency Response")
xlabel(r"$\omega$")
ylabel(r"$|H|$")
subplot(212)
semilogx(omega,180*angle(H)/pi,'ro')
xlabel(r"$\omega$")
ylabel(r"Arg$(H)$")
```



This shows the frequency response of the circuit both in magnitude and in phase. But it shows it as a function of ω. We are usually interested in the plot as a function of frequency. So here that is

5 $\langle$*eg1* 4$\rangle+\equiv$ $\triangleleft$4 6a$\triangleright$

```
figure(2)
cla()
f=omega/(2*pi)
subplot(211)
loglog(f,abs(H),'ro')
title("Frequency Response")
xlabel(r"$f\;(Hz)$")
ylabel(r"$|H|$")
subplot(212)
semilogx(f,180*angle(H)/pi,'ro')
xlabel(r"$f\;(Hz)$")
ylabel(r"Arg$(H)$")
```



We would like to know the poles of this transfer function. For that we find the roots of the denominator polynomial.

6a ⟨*eg1* 4⟩+≡ ◁5 6b▷

```
den=poly1d([L*C,R*C,1.0]) # s^{2}LC+sRC+1

print roots(den)
```

```
  Python: [ -9.99899990e+07  -1.00010002e+04]
```

The roots are both real and in the left half plane. This is a stable, overdamped system. So its response to a step excitation should decay to the final value without oscillations. Let us simulate that.

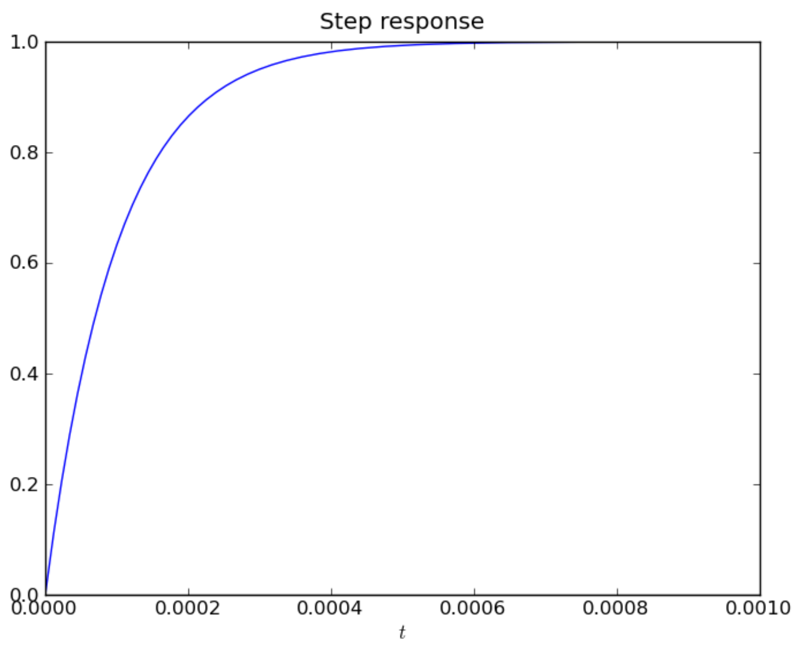
6b ⟨*eg1* 4⟩+≡ ◁6a 6c▷

```
from scipy import signal
sys = signal.lti(1,den)
```

This has created the transfer function $1/\left(s^2LC+sRC+1\right)$. We now simulate its response to a step input, $V_i = u_0(t)$:

6c ⟨*eg1* 4⟩+≡ ◁6b 7▷

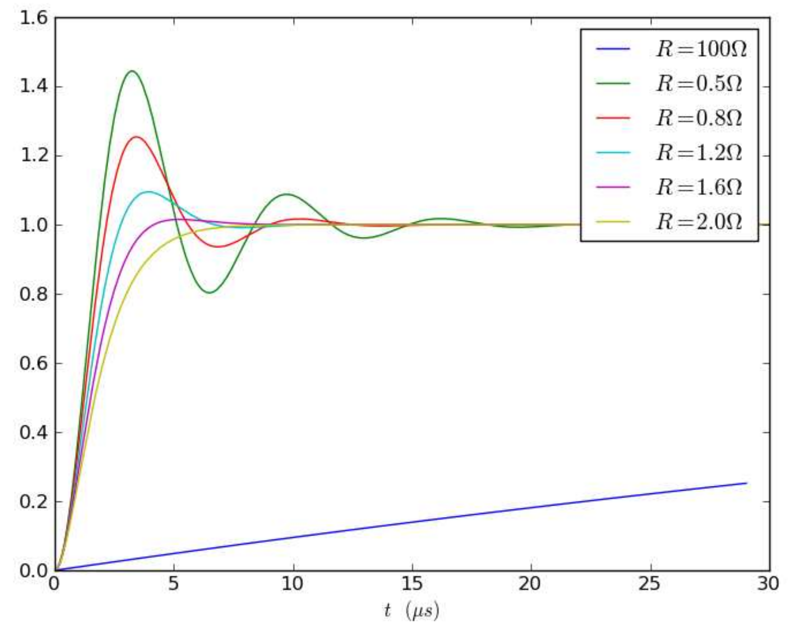
```
time=linspace(0,1e-3,1000)
step_response = sys.step(X0=[0,0],T=time)[1]
figure(3)
plot(time, step_response)
axhline(0, color='black')
xlabel(r'$t$')
title('Step response')
```



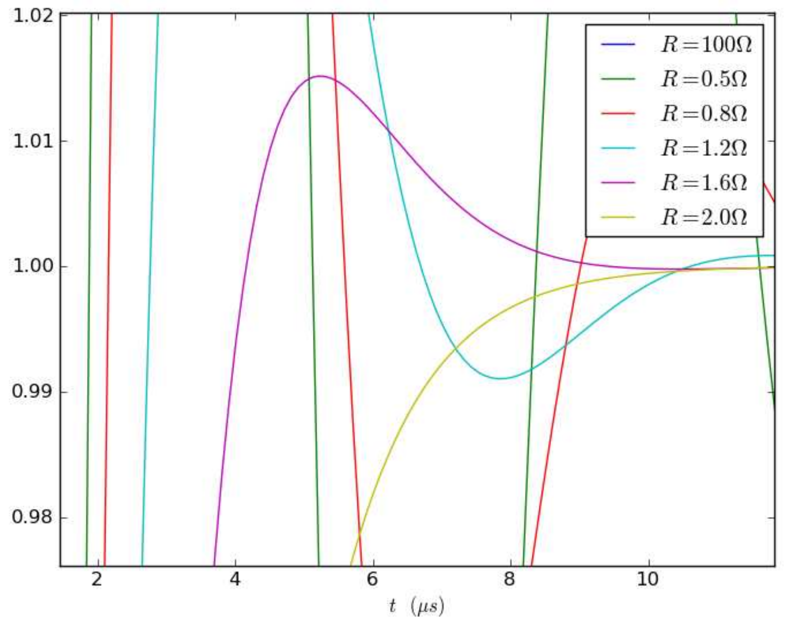
Now let us reduce the resistance from 100Ω to below 2Ω, which is the critical damping value.

7 ⟨*eg1* 4⟩+≡ ◁6c 9▷

```
Rvals=[0.5,0.8,1.2,1.6,2.01]
time1=linspace(0,3e-5,300)
figure(4)
cla()
ii=where(time<=3e-5) # clip the previous plot
plot(1e6*time[ii],step_response[ii])
str=[r"$R=100\Omega$"]
for R in Rvals:
  den1=poly1d([L*C,R*C,1])
  sys1=signal.lti(1,den1)
  step_response1 = sys1.step(X0=[0,0],T=time1)[1]
  plot(1e6*time1,abs(step_response1))
  str.append(r"$R=%.1f\Omega$" % R)
xlabel(r"$t\;(\mu s)$")
legend(str)
```



All of these plots reach close to the asymptotic value of unity much faster than the $R = 100\Omega$ case (the blue line at the bottom of the graph). Zooming into the plot,



you can see that if we require the output to reach its final value to within 2% as quickly as possible, $R = 1.6\Omega$ is better than $R = 2\Omega$. That is why, as you will learn in Control Engineering, we design our control dynamics to be slightly underdamped rather than critically damped to get the fastest response.

Now I want to get the response to an AC input,

$$v_i(t) = \sin(\omega_0 t)\, u_0(t)$$

This has a Laplace transform given by

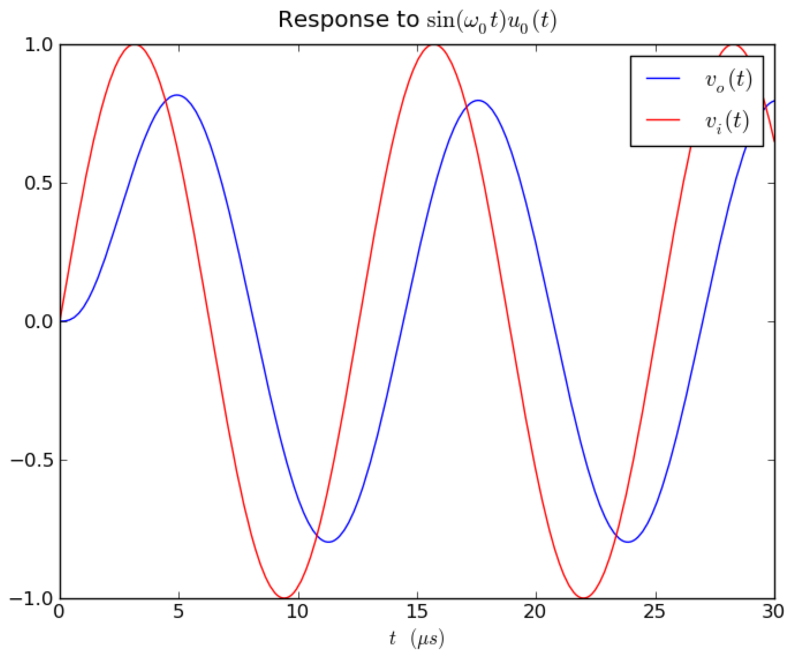
$$V_i(s) = \frac{\omega_0}{s^2+\omega_0^2} = \frac{\omega_0 s}{s^2+\omega_0^2} \times \frac{1}{s}$$

Thus, it is the same as the step response of

$$\frac{1}{s^2LC+sRC+1}\frac{\omega_0 s}{s^2+\omega_0^2}$$

9 $\langle$*eg1* 4$\rangle$+≡ ◁7

```
R=2.01         # Critical damping
omega0=5e5     # below resonance but not by much
num2=poly1d([omega0,0]) # \omega_{0}s+0

den2=polymul(den1,poly1d([1,0,omega0*omega0]))
sys2 = signal.lti(num2,den2)
step_response2 = sys2.step(T=time1)[1]
figure(5)
plot(1e6*time1,step_response2)
plot(1e6*time1,sin(omega0*time1),'r')
xlabel(r"$t\;(\mu s)$")
title(r"Response to $\sin(\omega_0 t)u_0(t)$")
legend([r"$v_o(t)$",r"$v_i(t)$"])
```



Clearly, the response settles down to a sinusoidal input as quickly as it does to a step input.

As you know from the Networks course, when you have a differential equation that has a driving function (the source), the solution is a sum of the particular solution that captures the nature of the driving function, and a solution of the homogeneous equation that takes care of initial conditions. The transients are due to the decay of the solution of the homogeneous equation, and this decay has the same rate independent of the source shape. This is why the settling time is the same for both the step input and the sinusoidal input.

3 Fourier Analysis

Suppose we consider our RLC filter. Our input is not a step or a simple sine wave. Instead it is a complex waveform. How can we get the output $v_0(t)$?

We have

$$V_0(\omega) = H(\omega)V_i(\omega)$$

So we have a procedure to solve for $v_0(t)$:

- From $v_i(t)$ compute $V_i(\omega)$
- From $V_i(\omega)$ compute $V_0(\omega)$
- From $V_0(\omega)$ compute $v_o(t)$

The first and third steps can no longer be done symbolically. We use instead the numerical fourier transform. So the steps become

- From $v_i(t)$ get samples $v_i(t_j)$ [Sampling]
- From $v_i(t_j)$ compute $V_i(\omega_k)$ [DFT]
- From $V_i(\omega_k)$ compute $V_0(\omega_k)$
- From $V_0(\omega_k)$ compute $v_o(t_j)$ [IDFT]
- From $v_o(t_j)$ compute $v_0(t)$ [Sinc Interpolation]

This full machinary is what the ADSP course is about. You will learn how many samples are required, and how, given enough samples, $v_0(t)$ is exactly obtained through this procedure.

Let us do it for a simple case - a square pulse. This can be analytically done, since a square pulse is nothing but a positive step followed by a negative step. Our square pulse turns on at $t = -5\mu$s and turns off at $t = 5\mu$s. We create a time vector from -25μs to 25μs.

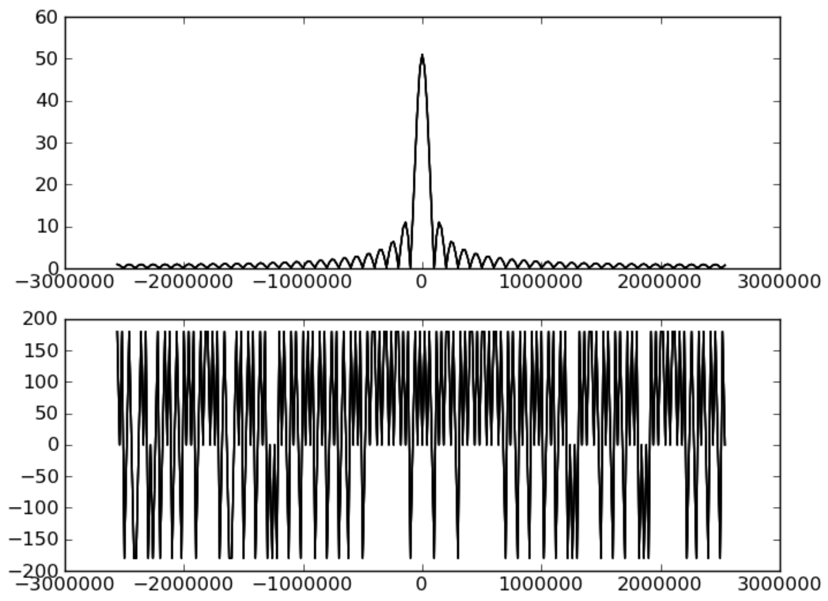
10 ⟨*eg3* 10⟩≡ 11a▷

```
from scipy import *
from matplotlib.pyplot import *
t=linspace(-25,25,257)*1e-6;t=t[0:-1]
vi=zeros(shape(t))
ii=where( (t>=-5e-6)*(t<=5e-6)>0 )
vi[ii]=1
figure(0)
cla()
plot(t,vi,'r')
```

We have the samples of the function $v_i(t)$. We get the numerical fourier transform using the "Fast Fourier Transform" or the FFT. This is a marvelous algorithm that computes the transform in a time that is proportional to $n \log n$.

11a ⟨*eg3* 10⟩+≡ ◁10 11b▷

```
Vi=fft(vi)
wmax=1/(t[1]-t[0])
frq=linspace(-0.5,0.5,257)*wmax
frq=frq[0:-1]
Vi=hstack([Vi[128:256],Vi[0:128]])   # fftshift
figure(1)
subplot(211)
plot(frq,abs(Vi),'k')
subplot(212)
plot(frq,angle(Vi)*180/pi,'k')
```



This very peculiar piece of code does two things. It computes the transform in `Vi`. It also creates the frequency vector. As you will learn in ADSP next semester, the frequency goes from 0 to $1/\Delta t$. However the $f = 1/\Delta t$ repeats the $f = 0$ component. So to get n different frequencies, we make $n+1$ and throw away the last.

Now our analog fourier transform goes from $-\infty$ to ∞ rather than from 0 to ∞. That is why I have the frequency vector defined that way. But the transform assumes frequency goes from 0 to ∞. So we have to shuffle the `Vi` vector as shown.

We now compute the output voltage. For that we need our Transfer Function. I have chosen $R = 1\Omega$ which is underdamped to make the graphs interesting.

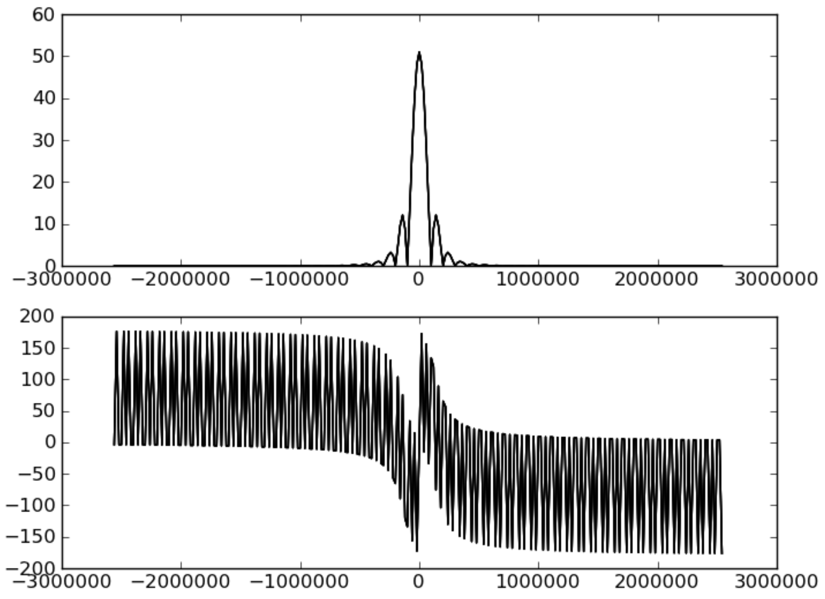
11b ⟨*eg3* 10⟩+≡ ◁11a 12a▷

```
omega=2*pi*frq
den=poly1d([-1e-12,1e-6j,1.0]) # -\omega^{2}LC+j\omega RC+1

H=1/den(omega)
```

The output is just the input multiplied by the Transfer Function.

12a ⟨*eg3* 10⟩+≡ ◁11b 12b▷

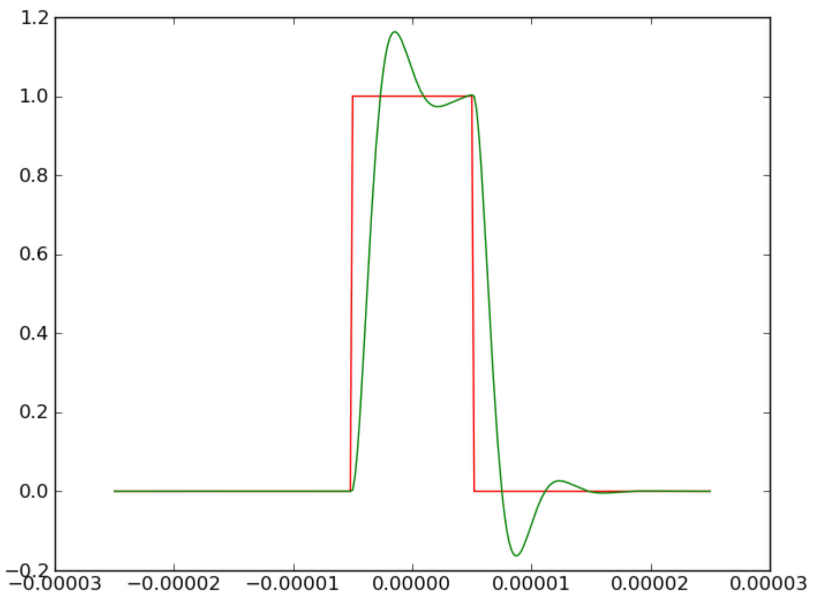
```
Vo=Vi*H
figure(2)
subplot(211)
plot(frq,abs(Vo),'k')
subplot(212)
plot(frq,angle(Vo)*180/pi,'k')
```



Finally we do the inverse transform to get the output voltage in time.

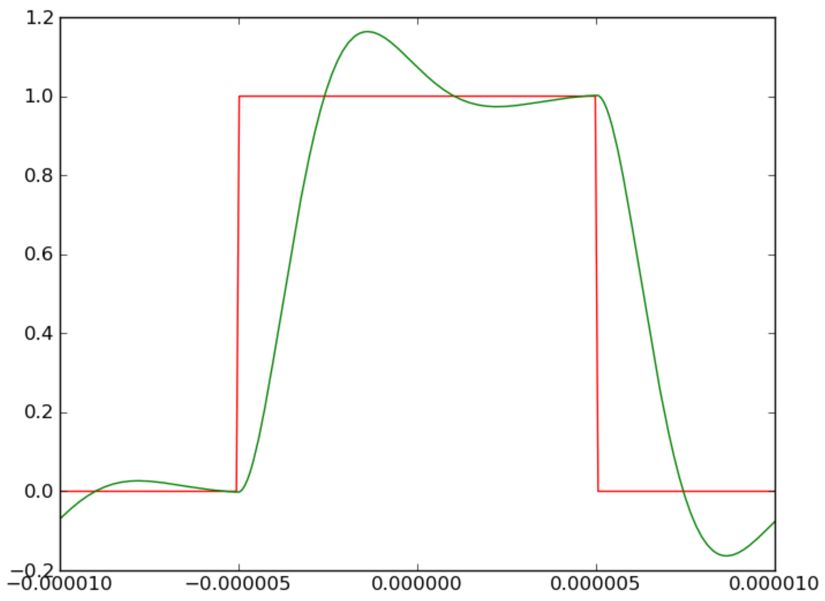
12b ⟨*eg3* 10⟩+≡ ◁12a

```
Vo=hstack([Vo[128:256],Vo[0:128]])   # fftshift
vo=ifft(Vo)
figure(0)
plot(t,vo,'g')
```



Look at the plot. It shows clearly what the effect of the underdamped transfer function does to the square wave input. The numerical Fourier Transform is perhaps the most important tool you will use in signal processing. Linear problems are far easier to do in transform space, and the `fft` and the `ifft` give us a very easy way to shift back and forth. But remember the little things we need to do to make it all work. The plot only plotted the real part. You can check and confirm that the imaginary part is negligible (due to round off error). If you did not take care as mentioned above, you will end up with complex `vo` in time, which is quite absurd.

There is a reason why we have chosen the time vector to be from -25μs to 25μs rather than, say, -10μs to 10μs. This is that the numerical fourier transform (also called the Discrete Fourier Transform or DFT) assumes that the function is periodic. So, if the output does not decay by the end of the time vector, *it continues into the negative end of the time vector!* So if we wish to simulate a non-periodic system with a DFT, we need to use a sufficiently large time vector so that all transients decay before the periodic constraint kicks in. Looking at the solution, we can see that transients are strongly present at 10μs, and this would completely spoil the solution. Below is what we get if we used $(-10\mu s, 10\mu s)$ as the time range. Clearly this is not the kind of simulation we want . . .



14 ⟨* 14⟩≡
```
⟨Q3 (never defined)⟩
show()
```