FFT analyis

The response function

The response function is

$$\frac{1}{1+2\zeta\beta_n i-\beta_n^2},\quad \beta_n = n\beta_1 = \frac{n\omega_1}{\omega_n},$$

with the caution that, for $n > N/2$, the frequency $n\omega_1$ must be wrapped.

Assuming that β_1, ζ and N are defined outside our function, we can write

In [23]:
```
def resfun(n):
    if n>N/2: n=n-N
    return 1.0/(1.0+n*b1*(2*z*1j-n*b1))
```

Note above that we have not introduced the dimensional factor $1/k$.

The load function

Just as in the spreadsheet:

In [24]:
```
def load(t):
    if t<t1: return p0*t
    if t<t2: return (1.5-0.5*t)*p0
    return 0.0
```

The loading data

We define the load over a period longer than its effective duration, we decide the number of samples and compute the fundamental frequency of the DFT.

In [25]:
```
p0=400000.0
t1=1.0
t2=3.0

T = 8.0
N = 4096

w1 = 2*pi/T
```

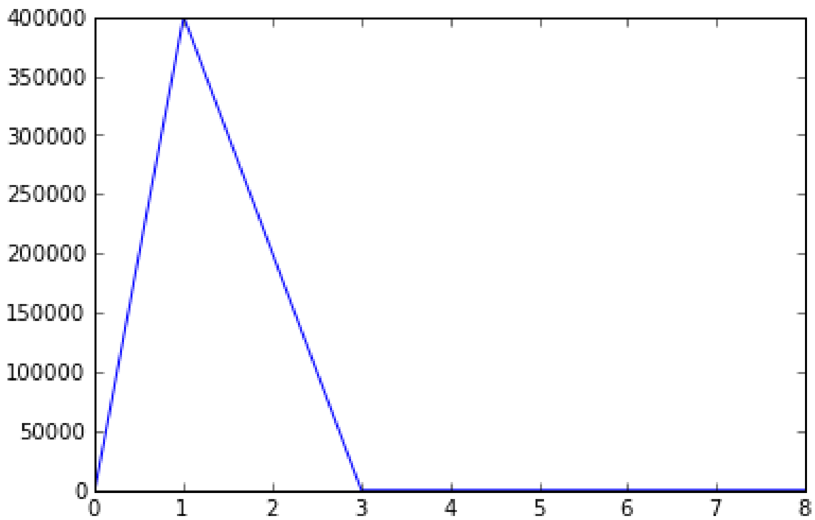
Generation of the load vector

First, the vector of times t_n, using a convenience function, then we apply the load function to this vector of times (the trick is `vectorize`-ing the load function, so that it can be applied to a vector.

Just to be sure that's all OK, let's plot the resulting load vector.

In [26]:
```
t=linspace(0., T, N, endpoint=False)
p=vectorize(load)(t)
figure(1); plot(t,p)
```

Out[26]: `[<matplotlib.lines.Line2D at 0x5660350>]`

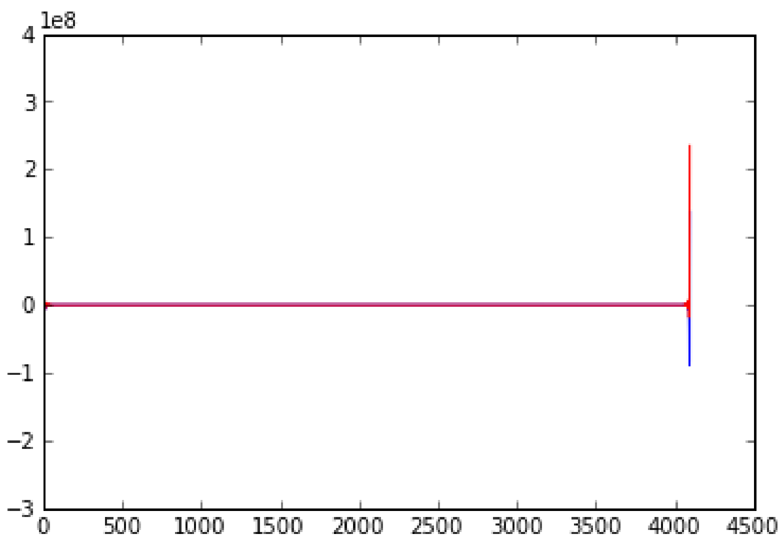


The FFT of the load

It's a line of code (let's plot the real and imaginary components of P.)

In [27]:
```
P = P = fft.fft(p+0j)
figure(2); plot(P.real,'-b', P.imag,'-r')
```

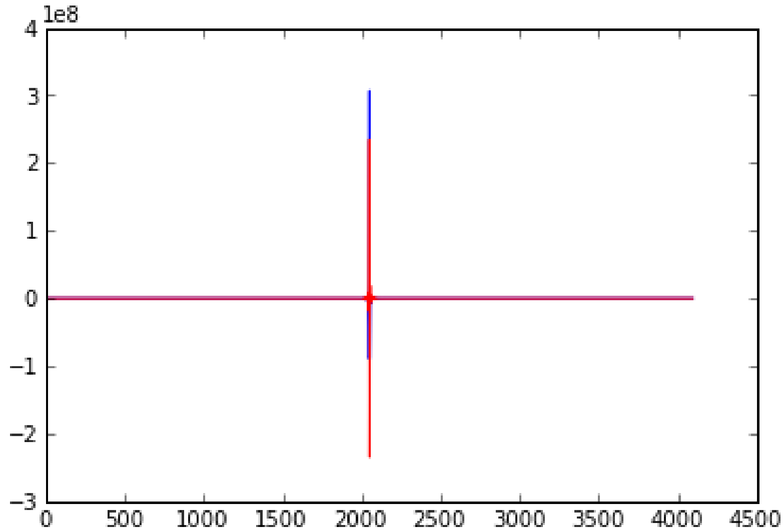
Out[27]:
```
[<matplotlib.lines.Line2D at 0x56986d0>,
 <matplotlib.lines.Line2D at 0x5698b90>]
```



Our plot is not very clear... there is a convenience fuction `fft.fftshift` that wraps the FFT placing the zero frequency element in the middle of the vector. We construct the shifted FFT and then we plot it:

In [28]:
```
Ps = fft.fftshift(P)
figure(3); plot(Ps.real,'-b', Ps.imag,'-r')
```

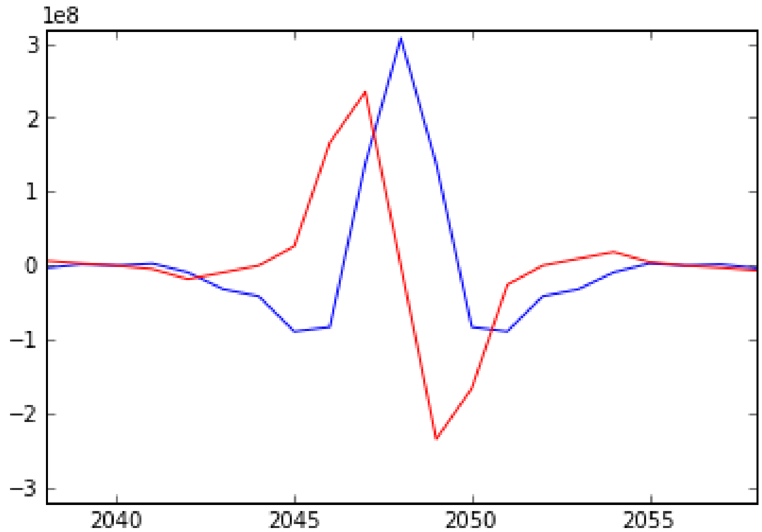
Out[28]:
```
[<matplotlib.lines.Line2D at 0x5919710>,
 <matplotlib.lines.Line2D at 0x5919bd0>]
```



It is however better to zoom the plot near the centre of the n axis

In [29]:
```
axis([2038,2058,-3.2e8,3.2e8]);plot(Ps.real,'-b', Ps.imag,'-r')
```

Out[29]:
```
[<matplotlib.lines.Line2D at 0x5921750>,
 <matplotlib.lines.Line2D at 0x5cb2690>]
```



Computing the response

The characteristics of the dynamic system

The mass, the natural period of vibration and the corresponding natural frequency, the damping ratio ζ and finally β_1, the frequency ratio associated with the fundamental frequency of the DFT of the loading

In [30]:
```
mass=60E3
Tn=0.60
wn = 2*pi/Tn ; k = mass*wn*wn
z=0.00
b1 = w1/wn
```

The FFT of the response

Is computed multiplying the DFT of the load by the vector with the samples of the response function, computed on the fly using the `vectorize` trick.

In [31]:
```
X=P*vectorize(resfun)(range(N))
```

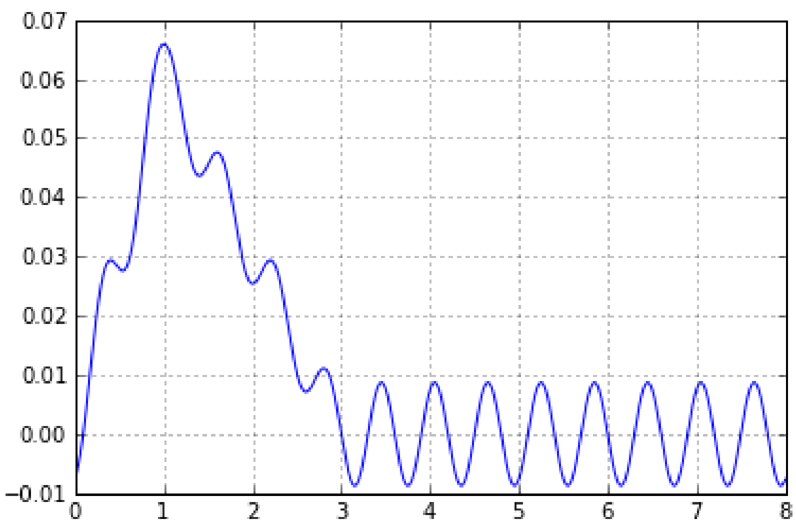
The response

Is computed applying the inverse DFT to the DFT of the response.

Then we plot it (applying the correction for static displacement) and look at what happens for $t = 0$

In [32]:
```
x=fft.ifft(X)
figure(4);grid();lot(t,x/k)plot(t,x/k)
```

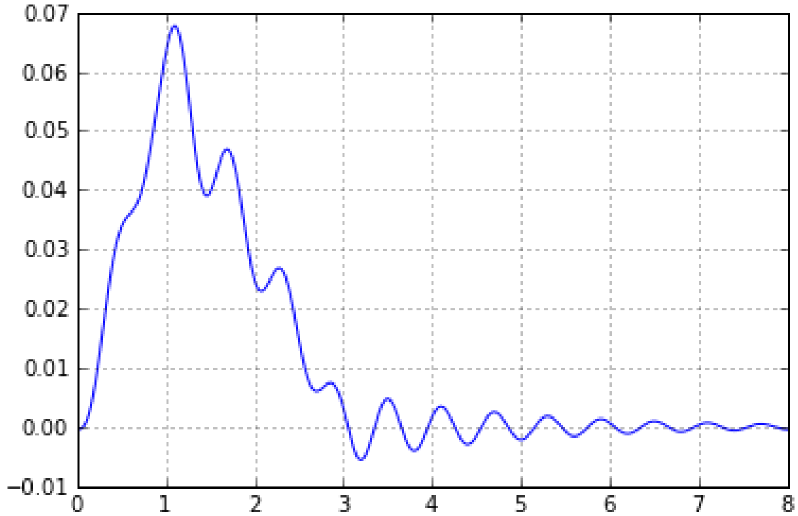
Out[32]:
```
[<matplotlib.lines.Line2D at 0x5c8fd10>]
```



As you can see, the initial conditions are different from (0,0). We change the damping ratio and compute again the response

In [33]:
```
z=0.05
X=P*vectorize(resfun)(range(N))
x=fft.ifft(X)
grid();plot(t,x/k)
```

```
Out[33]: [<matplotlib.lines.Line2D at 0x5f0e8d0>]
```



Now the initial conditions are respected with a *good approximation*.

The key point is, leave a *zero-trail* of sufficient length, so that the response at the end of the period is sufficiently close to zero.