Homework 1 Solutions

Here we have the homework #1 solutions, preceeded by a few imports from the (almost) standard library and the definition of an utility function.

In [1]:

```
import numpy as np
from numpy import cos, exp, pi, sin, sqrt

def pd(s, v, u=None):
    if u:
        print('%30s: %g [%s]'%(s, v, u))
    else:
        print('%30s: %g'%(s, v))
```

System Identification

The data, first as in text, next a bit of manipulation to have different values in different arrays

In [2]:
```
#       f    P      rho   theta
raw = [[18., 3240.,  54.,  24.3],
       [20., 4000., 118.,  55.1],
       [22., 4840., 132., 123.9],
       [24., 5760.,  80., 152.5]]
f, p, r, t = map(np.array, zip(*raw))

omega = f*2*pi
force = p
rho   = r/1E6
theta = t*pi/180.0
```

`A` is the matrix of coefficients and `b` the known term, `k` and `m` are computed using a least squares solver

In [3]:
```
A = np.vstack((np.ones(4), -omega**2)).T
b = force*cos(theta)/rho
k, m = np.linalg.lstsq(A, b, rcond=None)[0]
```

In [4]:
```
print('Matrix of coefficients,\n[ 1,  -omega^2_i ]')

print('\n'.join('[ %d,  %.4g ]'%(row[0],row[1]) for row in A))
print('Known term,  p_i·cos theta_i/rho_i,')
print('[', ', '.join('%.4g'%x for x in b), ']\n')
print('Stiffness [MN/m]', k/1E6)
print('      Mass [ton]', m/1000)
```

```
Matrix of coefficients,
[ 1,  -omega^2_i ]
[ 1,  -1.279e+04 ]
[ 1,  -1.579e+04 ]
[ 1,  -1.911e+04 ]
[ 1,  -2.274e+04 ]
Known term,  p_i·cos theta_i/rho_i,
[ 5.468e+07, 1.939e+07, -2.045e+07, -6.386e+07 ]

Stiffness [MN/m] 207.45835123684353
      Mass [ton] 11.927812957251826
```

First we print the four estimates of ζ from the four imprecise measurements, next the estimate obtained using the least squares solver.

In [5]:
```
wn2 = k/m
wn = sqrt(wn2)
beta = omega/wn
print('[', ', '.join('%.4g'%x for x in (force*sin(theta)/(2*rho*k*beta))), ']')
print(np.linalg.lstsq(np.ones((4,1)), force*sin(theta)/(2*rho*k*beta), rcond=None)[0][0
])
```

```
[ 0.06939, 0.07032, 0.06998, 0.07008 ]
0.06994247277376539
```

Let's say that $\zeta = 7\%$.

Vibration Isolation

The data, some easily derived quantity. `beta2` is the squared frequency ratio of the undamped, isolated system.

In [6]:

```
mass = 17.13E3
freq = 10.0
omega = freq*2*pi
TR = 1/3
beta2 = 1+1/TR
```

The frequency ratios ofthe damped systems are found using a library root solver,a bit of cheating isn't it?

Next, the stiffnesses for different dampings are $k = m\omega_n^2 = m\omega^2/\beta^2$.

In [7]:
```
from scipy.optimize import newton
def tr(b, z): return sqrt(1+4*b*b*z*z)/sqrt((1-b*b)**2+4*z*z*b*b)

b_01 = newton(lambda b: tr(b, 0.01)-TR, sqrt(beta2))
b_10 = newton(lambda b: tr(b, 0.10)-TR, sqrt(beta2))
k_00 = mass*omega**2/beta2
k_01 = mass*omega**2/b_01**2
k_10 = mass*omega**2/b_10**2
```

In [8]:

```
print("Damping Ratio,   Damping Coeff. [kN·s/m],    Stiffness [MN/m]")
print("           0%%, %25.3f, %19.3f"%(0.0, k_00/1E6))
print("           1%%, %25.3f, %19.3f"%(
    0.02*sqrt(k_01*mass)/1000, k_01/1E6))
print("          10%%, %25.3f, %19.3f"%(
    0.20*sqrt(k_10*mass)/1000, k_10/1E6))
```

```
Damping Ratio,   Damping Coeff. [kN·s/m],    Stiffness [MN/m]
           0%,                     0.000,              16.907
           1%,                    10.760,              16.898
          10%,                   104.824,              16.036
```

Impulsive Load + Num.Integration

The data of the problem and some easily derived parameters

In [9]:

```
mass = 400
wn = 2*pi*4
z = 0.03
p0 = 8200.0
t0 = 0.040

stif = mass*wn**2
damp = 2*z*wn*mass
Tn = 2*pi/wn

beta = Tn/2/t0
Dst = p0/stif
```

We have a formula for the maximum of free response for a half sine excitation...

In [10]:
```
Rmax = 2*beta*cos(pi/2/beta)/(beta**2-1)

xmax_00_exact = Dst*Rmax

print('Exact formula for undamped system')
pd('Static displacement', Dst*1000, 'mm')
pd('Max response coeff.', Rmax)
pd('Max displacement', xmax_00_exact*1000, 'mm')
```

```
Exact formula for undamped system
           Static displacement: 32.4544 [mm]
           Max response coeff.: 0.624818
              Max displacement: 20.2781 [mm]
```

Now the approximate formula, valid for every type of short impulse. The integral of the half sine is

$$p_0 \int_0^{t_0} sin(\pi\tau/t_0)\, d\tau = 2p_0 t_0/\pi$$

In [11]:
```
print('Impulse-momentum approximate result')
integral = p0*2*t0/pi
xmax_00_approx = integral/mass/wn

pd('Max displacement', xmax_00_approx*1000, 'mm')
```

```
Impulse-momentum approximate result
              Max displacement: 20.7708 [mm]
```

And the numerical solution (now we take into account the damping).

We choose a total duration, a time step, we instantiate a time vector and define the loading and the load increments.

In [12]:
```
t1 = 0.100
N = 1000
h = t1/N
t = np.linspace(0, t1, N+1)
p = p0*np.where(t<=t0, sin(pi*t/t0), 0.0)
Dp = p[1:]-p[:-1]
```

The constants for the *Constant Acceleration Algorithm* (they depend on h)

In [13]:
```
ks = stif + 2*damp/h + 4*mass/h/h
cs = 2*damp + 4*mass/h
ms = 2*mass
```

so that in the next slide we can compute the solution (up to the point of a velocity reversal).

In [14]:
```
x0, v0 = 0, 0
for tt, p0, dp in zip(t, p, Dp):
    a0 = (p0-damp*v0-stif*x0)/mass
    dps = dp+ms*a0+cs*v0
    dx = dps/ks
    dv = 2*(dx/h-v0)
    x0, v0 = x0+dx, v0+dv
    if v0<0: break

pd('Max displacement', x0*1000, 'mm')
```

```
Max displacement: 19.3616 [mm]
```

Rayleigh Quotient and Refinements

The data of the problem, in order the assumed displacements, the floor masses and the storey stiffnesses. We define also a fictitious frequency and its square and eventually we import the `Fraction` class from the standard library for a later use.

In [15]:
```
x0, x1, x2, x3 = 0, 1, 2, 3
m1, m2, m3 = 5, 5, 3
k1, k2, k3 = 8, 5, 2

w = 1 ; w2 = w*w

from fractions import Fraction as f
```

To compute the (double of the) strain energy we need the storey deflections, `d1` etc.

The RQ is simply the fraction (`f`, that is) with `V2` in the numerator and `T2` in the denominator.

In [16]:

```
d1, d2, d3 = x1-x0, x2-x1, x3-x2
V2 = k1*d1**2 + k2*d2**2 + k3*d3**2
T2 = w**2 * (m1*x1**2 + m2*x2**2 + m3*x3**2)

R00 = f(V2,T2)
```

To proceed with refinements we need the inertial forces (`f1`, etc), the storey shears (`v1`, etc, note that we have to sum the floor forces from the top to the bottom), the storey deflections (`d1`, etc, computed as exact fractions using `f`) and eventually the floor displacements (`x1`, etc, this time we sum from the bottom to the top).

In [17]:
```
f1, f2, f3 = w2*m1*x1, w2*m2*x2, w2*m3*x3
print("Inertial forces, f_i/(m w**2)", f1, f2, f3)

v3 = f3 ; v2 = v3+f2 ; v1 = v2+f1
print("Storey shears, F_i/(m*w**2)  ", v1, v2, v3)

d1, d2, d3 = f(v1, k1), f(v2, k2), f(v3, k3)
print("Storey deflections, d_i*k/(m*w**2)", d1, d2, d3)

x1 = x0+d1 ; x2 = x1+d2 ; x3 = x2+d3
print("Storey displacements, x_i*k/(m*w**2)", x1, x2, x3)
```

```
Inertial forces, f_i/(m w**2) 5 10 9
Storey shears, F_i/(m*w**2)   24 19 9
Storey deflections, d_i*k/(m*w**2) 3 19/5 9/2
Storey displacements, x_i*k/(m*w**2) 3 34/5 113/10
```

With the new displacements and the old forces, estimate a better V2 and next compute R01 and eventually a better kinetic energy T2 and R11.

In [18]:
```
V2 = f1*x1 + f2*x2 + f3*x3
R01 = f(T2, V2)

T2 = w2*( m1*x1**2 + m2*x2**2 + m3*x3**2)
R11 = f(V2, T2)
```

It's time to display our results

In [19]:
```
def p_Rxx(Rs, Rv):
    print('%10s = %15s k/m = %g k/m.'%(Rs, str(Rv), 1.0*Rv))

p_Rxx('R00', R00)
p_Rxx('R01', R01)
p_Rxx('R11', R11)
```

```
       R00 =           15/52 k/m = 0.288462 k/m.
       R01 =         520/1847 k/m = 0.281538 k/m.
       R11 =      18470/65927 k/m = 0.280158 k/m.
```