Exact integration of the equation of motion for an elastoplastic oscillator

April 5, 2011

We consider an oscillator with mass $m = 1000$kg, stiffness $k = 40000$kN/m and a damping ratio $\zeta = 3\%$ and yielding limit $f_y = 2500$N, loaded by a half-sine impulse,

$$p(t) = \begin{cases} P_0 \sin(\pi t/t_1), & 0 \le t \le t_1, \\ 0 & \text{otherwise} \end{cases}$$

where $P_0 = 6000$N and $t_1 = 0.3$s.

In this paper we discuss a computer program written to integrate the equation of motion of the particular system described above, where I say *particular system* because the execution branches are taken by prior knowledge of the behaviour of the system, and not using tests in a step-by step fashion, as it should be in a general solution.

The code, denoted by the different background, is written in the programming language python.

python is a very simple programming language, so simple that lots of stuff are not directly available but reside in external libraries or *modules*. The standard libraries, on the other hand, cover a large spectrum of necessities and there are also a great number of third party modules (last week we used a third party library, the pylab module)

When we need something outside the base language, we have to know the module that provides our requirements and *import* some name from it. In our case, we need the usual mathematical functions, so we start yhe program with the statement

```
from math import *
```

Importing the asterisk from the module math means to import*everything*, that is all names that are defined in the module; most names refer to functions, like sin or cos, but in math are defined also constants like pi $= \pi$.

It is customary to write first all the imports, to make apparent what external modules the program requires; the form we used for our import was not as idiomatic as **from** math **import** sin, cos, exp,sqrt, pi that clarifies exactly what names we have imported from the math module.

Another variation on import, the most explicit one..., is **import** math and in this case we must prefix every references to the objects from math, as in pi2 = math.pi*2 in this case it becomes very simple to asses the origin of any object that is used in our code.

After this aside on importing objects' names in our code, we define a function that returns two functions, namely the elastic displacement and the elastic velocity of a sdof, subjected to an assigned load and given initial conditions.

```
def resp_elas(m,c,k, cC,cS,w, F, x0,v0):
```

where m,c,k are the SDOF's characteristics, x0,v0 are the initial conditions and cC,cS,w, F define the loading, $p(t) = \text{cC}\cos(\text{w}t) + \text{cS}\sin(\text{w}t) + F$. Note that cC and cS must be *forces*, as well as the constant force F

```
    wn2=k/m ; wn=sqrt(wn2) ; beta=w/wn
    z=c/(2*m*wn) ; wd=wn*sqrt(1-z*z)
```

After computing the dynamic parameters, we compute the coefficients in the particular integral

```
    # csi(t) = R sin(w t) + S cos(w t) + D
    det=(1.-beta**2)**2+(2*beta*z)**2
    R=((1-beta**2)*cS + (2*beta*z)*cC)/det/k
    S=((1-beta**2)*cC - (2*beta*z)*cS)/det/k
    D=F/k
```

Now the constants in the general integral, using the initial conditions,

```
    # x(0) = 1 * ( A*1 + B*0 ) + R*0 + S*1 + D = x0
    A=x0-S-D
    # v(0) = wd B - z wn A + w R = v0
    B=(v0+z*wn*A-w*R)/wd
```

and using the general and the particular integral constants, we can define the two functions that compute the response (displacement and velocity) for $t > 0$.

```
    def x(t):
      return (exp(-z*wn*t)*(A*cos(wd*t)+B*sin(wd*t))
              +R*sin(w*t)+S*cos(w*t)+D)
    def v(t):
      return (-z*wn*exp(-z*wn*t)*(A*cos(wd*t)+B*sin(wd*t))
              +wd*exp(-z*wn*t)*(B*cos(wd*t)-A*sin(wd*t))
              +w*(R*cos(w*t)-S*sin(w*t)))
```

Finally, we return to the caller these two functions! In python, functions are objects like integers, floats, and other types of variables, and can be binded to a name, e.g., s=sin; **print** s(3.14/2) prints 1.0.

```
    return x,v
```

Next, we define another function defining functions, that returns the displacement and velocity during the yielding phase. Examining the code you could take a guess at the particular and general integral to the equation $m\ddot{x} + c\dot{x} = \text{cC}\cos(\text{w}t) + \text{cS}\sin(\text{w}t) + F$.

```
def resp_yield(m,c, cC,cS,w, F, x0,v0):
    # csi(t) = R sin(w t) + S cos(w t) + \alpha t
    # x(t) = A exp(-c t/m) + B +
    #          + R sin(w t) + S cos(w t) + alpha t
    # v(t) = - c A/m exp(-c t/m) +
    #          + w R cos(w t) - w S sin(w t) + alpha
    alpha=F/c
    det=w**2*(c**2+w**2*m**2)
    R=(+w*c*cC-w*w*m*cS)/det ; S=(-w*c*cS-w*w*m*cC)/det
    # v(0) = -c A / m + w R + alpha = v0
    A=m*(w*R+alpha-v0)/c
    # x(0) = A + B + S = x0
    B=x0-A-S
    def x(t):
        return ( A*exp(-c*t/m) + B
                 + R*sin(w*t) + S*cos(w*t) + alpha*t )
    def v(t):
        return ( -c*A*exp(-c*t/m)/m
                   + w*R*cos(w*t) - w*S*sin(w*t) + alpha )
    return x,v
```

The next function we're going to define is a helper function, that returns a root t^* using the simple method of bisection, $f(t^*) = f_0$:

```
def bisect(f, root, x0, x1):
    h = (x0 + x1)/2.0
    fh = f(h) - root
    if abs(fh) < 1e-8 : return h
    f0 = f(x0) - root
    if f0*fh > 0:
        return bisect(f, root, h, x1)
    else:
        return bisect(f, root, x0, h)
```

The last block is about output, a sad necessity in most programs... we define a function to tabulate a function f$(t - t_1)$ over an interval t_1 to t_2 using $n+1$ points:

```
def tabulate(f, t_1, t_2, n_steps):
     step = (t_2 - t_1)/n_steps
     for i in range(n_steps+1):
        tau = i*step
        print t_1+tau, f(tau)
    return
```

Having defined all the building blocks, we set the sdof parameters,

```
mass=1000. # kg
k=40000.   # N/m
zeta=0.03  # damping ratio
damp=2*zeta*mass*sqrt(k/mass)
fy=2500.   # N,  the yielding force in the spring
xy=fy/k    # m,  the displacement of 1st yield
```

and the characteristics of the loading

```
t1=0.3     # s
w=pi/t1    # rad/s
Po=6000.   # N
```

To compute the response functions and start our computation we need the initial elastic response functions and, with null initial conditions, we set

```
x0=0.0     # m
v0=0.0     # m/s
x_next,v_next=resp_elas(mass,damp,k, 0.0,Po,w, 0.0, x0,v0)
```

Before starting, we compute the time t_y for which the yield take place,

```
t_yield=bisect(x_next,xy,0.0,t1,1)
```

Because $t_y < t_1$, I decided the time interval $(0, t_y)$ where the response is linear. To repeat the concept, in every program it is sadly necessary to generate some output. Here I print 101 time-displacement points between the start of excitation and yielding.

```
tabulate(x_next, 0.0, t_yeld, 100)
```

At this point, $t = t_y$, the spring is yielding: we find the new initial conditions,

```
x0 = x_next(t_y - 0.0); v0 = v_next(t_y - 0.0)
```

and the new load coefficients, with $\tau = t - t_y$ we have that $p(\tau) = (\cos(\omega t_y)\sin(\tau) + \sin(\omega t_y)\cos(\tau))P_0$

```
cS = cos(w*t_y)*Po
cC = sin(w*t_y)*Po
```

and the new response functions.

```
x_next, v_next = resp_yield(mass,damp,    cC,cS,w, -fy, x0,v0)
```

Note that the constant force in the function call above is opposite to the yielding force, as the yielded spring continues to exert the yielding force on the mass.

The upper limit of validity of this response is the smaller time between t_1, where the load changes, and $t_{v=0}$, where we could go back in the elastic phase. In this case, the interval of validity is (t_y, t_1), as I found by inspection.

Let's print some points in this interval,

```
tabulate(x_next, t_yeld, t_1, 100)
```

Now, $t = t_1$ and $p \equiv 0$, we must change the response functions because the external load changed.

```
cS = 0.0 ; cC = 0.0
x0 = x_next(t_1 - t_y)
v0 = v_next(t_1 - t_y)

x_next, v_next = resp_yield(mass, damp, cC, cS, w, -fy, x0, v0)
```

Now, the spring is yielded and the velocity is positive, we'll remain in this yielding phase until the velocity equals zero, so we find this phase change time

```
t2=t1+bisect(v_next, 0.0, 0, 0.3, 1.0)
```

having found t_2, we print 101 points in this interval

```
tabulate(x_next, t_1, t_2, 100)
```

Now the velocity is 0.0, going back to elastic behaviour... note the use of a constant force to model the permanent displacement.

```
x0 = x_next(t_2 - t_1) ; v0 = 0.0
x_next, v_next = resp_elas(mass,damp,k, 0.0,0.0,w, k*x0-fy, x0,v0)
```

Finally, we print some points following the return of the sdof in the elastic phase.

```
tabulate(x_next, t_2, 4.0, 200)
```

A similar program can be written to compute the indefinitely elastic response, the results are in the following figures, the first one shows in more detail the yielding phase, highlighting $t_y \approx 0.203$s and $x_y = 0.0625$cm, the time and displacement of fist yielding, the second one highlighting the differences in the free response, namely *a*) the permanent yielding displacement and *b*) the different amplitudes of the vibrations, associated with the higher dissipation of energy that takes place in the yielded oscillator.