Exact Integration for an EP SDOF System

We want to compute the response, using the constant acceleration algorithm plus MNR, of an Elasto Plastic (EP) system... but how we can confirm or reject our results?

It turns out that computing the exact response of an EP system with a single degree of freedom is relatively simple.

Here we discuss a program that computes the analytical solution of our problem.

The main building blocks of the program will be two functions that compute, for the elastic phase and for the plastic phase, the analytical functions that give the displacement and the velocity as functions of time.

Elastic response

We are definining a function that, for a linear dynamic system, returns not the displacement or the velocity at a given time, but rather a couple of functions of time that we can use afterwards to compute displacements and velecities at any time of interest.

The response depends on the parameters of the dynamic system $m, c, k,$ on the initial conditions $x_0, v_0,$ and on the characteristics of the external load.

Here the external load is limited to a linear combination of a cosine modulated, a sine modulated (both with the same frequency ω) and a constant force,

$$P(t) = c_C \cos\omega t + c_S \sin\omega t + F,$$

but that's all that is needed for the present problem.

The particular integral being

$$\xi(t) = S\cos\omega t + R\sin\omega t + D,$$

substituting in the equation of motion and equating all the corresponding terms gives the undetermined coefficients in $\xi(t)$, then evaluation of the general integral and its time derivative for $t = 0$ permits to find the constants in the homogeneous part of the integral.

The final step is to define the displacement and the velocity function, according to the constants we have determined, and to return these two function to the caller

In [1]:
```
def resp_elas(m,c,k, cC,cS,w, F, x0,v0):
    wn2 = k/m ; wn = sqrt(wn2) ; beta = w/wn
    z = c/(2*m*wn)
    wd = wn*sqrt(1-z*z)
    # xi(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
    A = x0-S-D
    B = (v0+z*wn*A-w*R)/wd

    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)))
    return x,v
```

Plastic response

In this case the equation of motion is

$$m\ddot{x} + c\dot{x} = P(t),$$

the homogeneous response is

$$x(t) = A\exp(-\frac{c}{m}t) + B,$$

and the particular integral, for a load described as in the previous case, is again

$$\xi(t) = S\cos\omega t + R\sin\omega t + D.$$

Having computed R, S, and D from substituting ξ in the equation of motion, A and B by imposing the initial conditions,we can define the displacement and velocity functions and, finally, return these two functions to the caller.

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

    return x,v
```

An utility function

We need to find when

I. the spring yields
II. the velocity is zero

to individuate the three ranges of different behaviour

I. elastic
II. plastic
III. elastic, with permanent deformation.

We can use the simple and robust algorithm of *bisection* to find the roots for

$$x_{el}(t) = x_y \text{ and } \dot{x}_{ep}(t) = 0.$$

In [3]:
```
def bisect(f,val,x0,x1):
    h  = (x0+x1)/2.0
    fh = f(h)-val
    if abs(fh)<1e-8 : return h
    f0 = f(x0)-val
    if f0*fh > 0 :
        return bisect(f, val, h, x1)
    else:
        return bisect(f, val, x0, h)
```

The system parameters

In [4]:
```
mass = 1000.   # kg
k    = 40000.  # N/m
zeta = 0.03    # damping ratio
fy   = 2500.   # N
```

Derived quantities

The damping coefficient c and the first yielding displacement, x_y.

In [5]:
```
damp = 2*zeta*sqrt(k*mass)
xy = fy/k    # m
```

Load definition

Our load is a half-sine impulse

$$p(t)=\begin{cases} p_0\sin(\frac{\pi t}{t_1}) & 0\leq t\leq t_1,\\ 0 & \text{otherwise.}\end{cases}$$

In our exercise

In [6]:
```
t1 = 0.3     # s
w  = pi/t1   # rad/s
Po = 6000.   # N
```

The actual computations

Elastic, initial conditions, get system functions

In [7]:
```
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)
```

Yielding time is

The time of yielding is found solving the equation $x_{\text{next}}(t) = x_y$

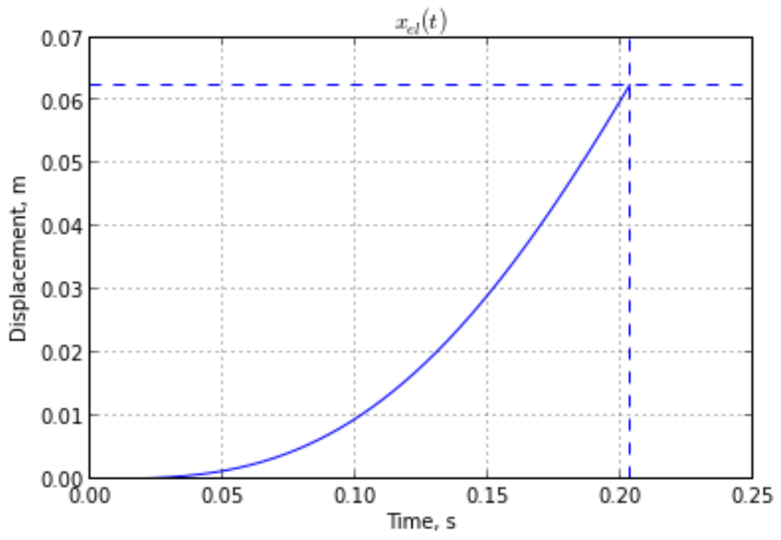
In [8]:
```
t_yield = bisect(x_next, xy, 0.0, t1)
print t_yield, x_next(t_yield)*k
```

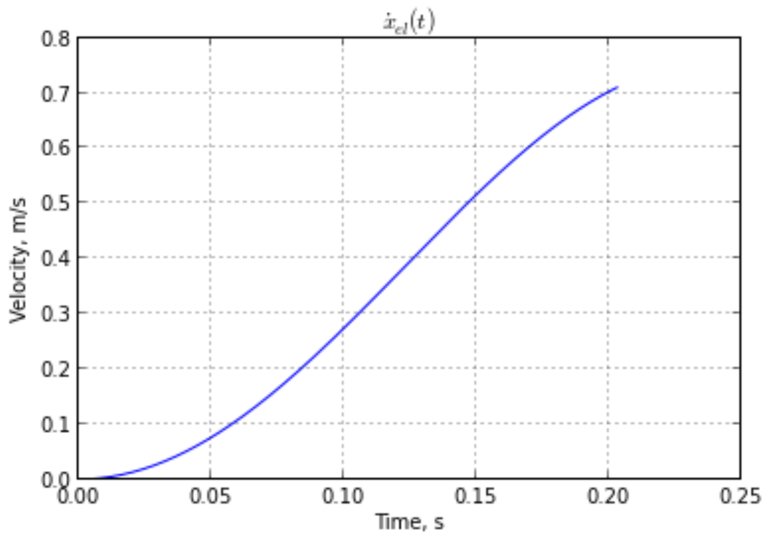
```
0.203265702724 2500.00009219
```

Forced response in elastic range is

In [9]:
```
t_el = linspace( 0.0, t_yield, 201)
x_el = vectorize(x_next)(t_el)
v_el = vectorize(v_next)(t_el)
# ------------------------------
figure(0);grid()
plot(t_el,x_el,
     (0,0.25),(xy,xy),'--b',
     (t_yield,t_yield),(0,0.0699),'--b')
title("$x_{el}(t)$")
xlabel("Time, s")
ylabel("Displacement, m")
# ------------------------------
figure(1);grid()
plot(t_el,v_el)
title("$\dot x_{el}(t)$")
xlabel("Time, s")
ylabel("Velocity, m/s")
```

Out[9]:
```
<matplotlib.text.Text at 0x7f722011aa10>
```





Preparing for EP response

First, the system state at t_y is the initial condition for the EP response

In [10]:
```
x0=x_next(t_yield)
v0=v_next(t_yield)
print x0, v0
```

```
0.0625000023047 0.709743249699
```

now, the load must be expressed in function of a *restarted time*,

$$\tau = t - t_y \;\rightarrow\; t = \tau + t_y \;\rightarrow\; \sin(\omega t) = \sin(\omega\tau + \omega t_y)$$

$$\rightarrow\; \sin(\omega t) = \sin(\omega\tau)\cos(\omega t_y) + \cos(\omega\tau)\sin(\omega t_y)$$

In [11]:
```
cS = Po*cos(w*t_yield)
cC = Po*sin(w*t_yield)

print Po*sin(w*0.55), cS*sin(w*(0.55-t_yield))+cC*cos(w*(0.55-t_yield))
```

```
-3000.0 -3000.0
```

Now we generate the displacement and velocity functions for the yielded phase, please note that the yielded spring still exerts a constant force f_y on the mass, and that this fact must be (and it is) taken into account.

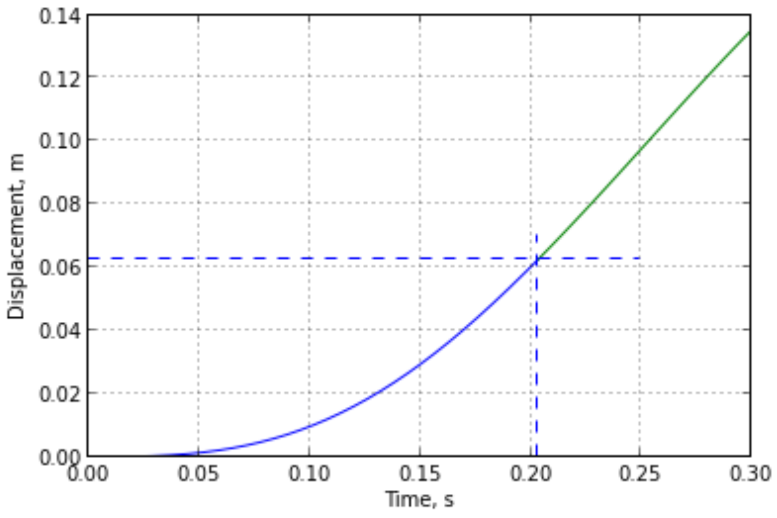
In [12]:
```
x_next, v_next = resp_yield(mass, damp,    cC,cS,w, -fy, x0,v0)
```

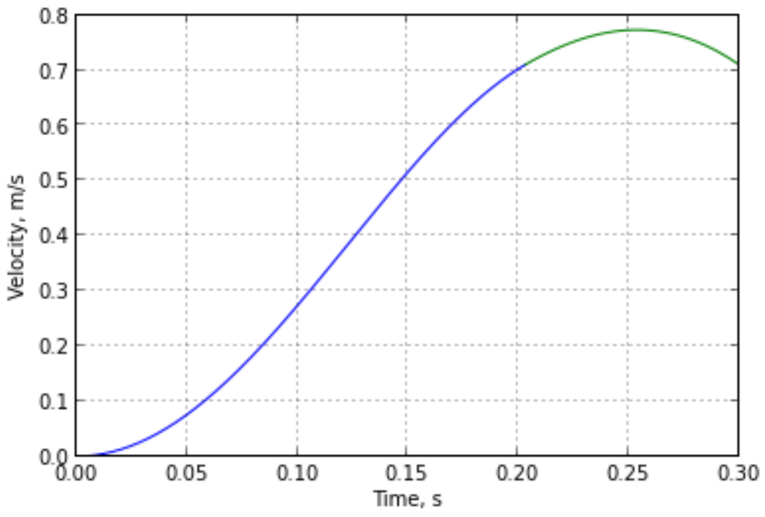
At this point I must confess that I have already peeked the numerical solution, hence I know that the velocity at $t = t_1$ is grater than 0 and I know that the current solution is valid in the interval $t_y \leq t \leq t_1$.

In [13]:
```
t_y1 = linspace(t_yield, t1, 101)
x_y1 = vectorize(x_next)(t_y1-t_yield)
v_y1 = vectorize(v_next)(t_y1-t_yield)
```

```
In [14]: figure(3) ; grid()
         plot(t_el,x_el, t_y1,x_y1,
              (0,0.25),(xy,xy),'--b',
              (t_yield,t_yield),(0,0.0699),'--b')
         xlabel("Time, s")
         ylabel("Displacement, m")
         # -----------------------------
         figure(4) ; grid()
         plot(t_el, v_el,  t_y1, v_y1)
         xlabel("Time, s")
         ylabel("Velocity, m/s")
```

```
Out[14]: <matplotlib.text.Text at 0x7f72202a9bd0>
```



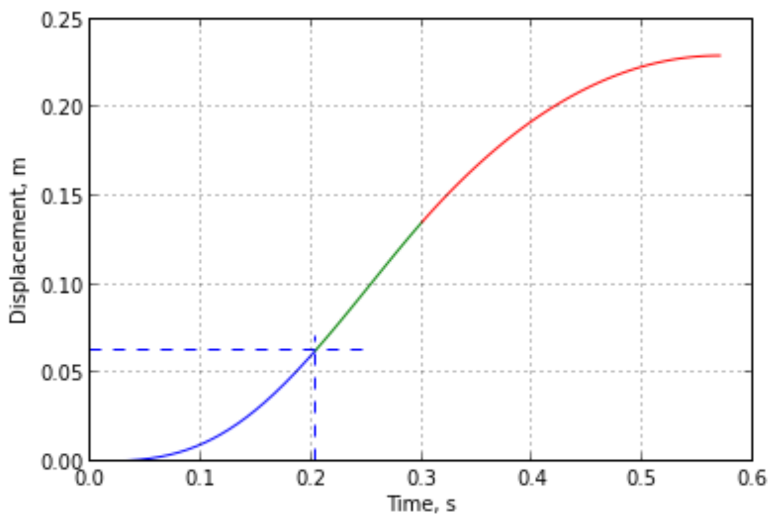


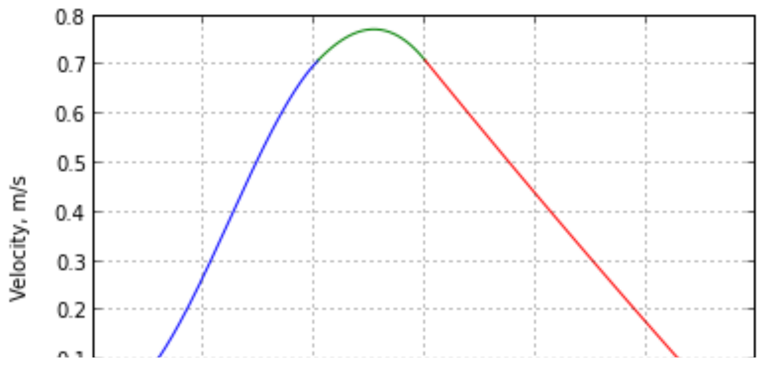
In the next phase, still it is $\dot{x} > 0$ so that the spring is still yielding, but now $p(t) = 0$, so we must compute two new state functions, starting as usual from the initial conditions (note that the yielding force is still applied)

In [15]:
```
x0 = x_next(t1-t_yield)
v0 = v_next(t1-t_yield)
print x0, v0
x_next, v_next = resp_yield(mass, damp, 0, 0, w, -fy, x0, v0)

t2 = t1 + bisect( v_next, 0.0, 0, 0.3)
print t2
t_y2 = linspace( t1, t2, 101)
x_y2 = vectorize(x_next)(t_y2-t1)
v_y2 = vectorize(v_next)(t_y2-t1)
print x_next(t2-t1)
# -------------------------------
figure(5) ; grid()
plot(t_el,x_el, t_y1,x_y1, t_y2, x_y2,
     (0,0.25),(xy,xy),'--b',
     (t_yield,t_yield),(0,0.0699),'--b')
xlabel("Time, s")
ylabel("Displacement, m")
# -------------------------------
figure(6) ; grid()
plot(t_el, v_el,  t_y1, v_y1, t_y2, v_y2)
xlabel("Time, s")
ylabel("Velocity, m/s")
```

```
0.135209330223 0.709996878577
0.569713139534
0.229324078054
```

Out[15]:
```
<matplotlib.text.Text at 0x7f72207cbd50>
```





Elastic unloading

The only point worth commenting is the constant force that we apply to our system.

The force-displacement relationship for an EP spring is

$$f_\mathrm{E} = k(x - x_\mathrm{pl}) = kx - k(x_\mathrm{max} - x_y)$$

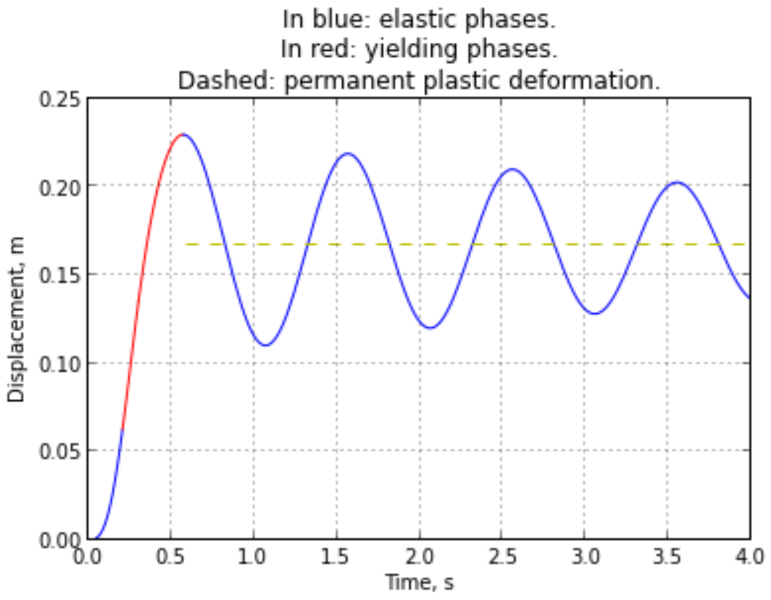
taking the negative, constant part of the last expression into the right member of the equation of equilibrium we have a constant term, as follows

```
In [16]: x0 = x_next(t2-t1) ; v0 = 0.0
         x_next, v_next = resp_elas(mass,damp,k, 0.0,0.0,w, k*x0-fy, x0,v0)
         t_e2 = linspace(t2,4.0,201)
         x_e2 = vectorize(x_next)(t_e2-t2)
         v_e2 = vectorize(v_next)(t_e2-t2)
```

now we are ready to plot the whole response

```
In [17]: # ------------------------------
         figure(7) ; grid()
         plot(t_el, x_el, '-b',
              t_y1, x_y1, '-r',
              t_y2, x_y2, '-r',
              t_e2, x_e2, '-b',
              (0.6, 4.0), (x0-xy, x0-xy), '--y')
         title("In blue: elastic phases.\n"+
               "In red: yielding phases.\n"+
               "Dashed: permanent plastic deformation.")
         xlabel("Time, s")
         ylabel("Displacement, m")
```

```
Out[17]: <matplotlib.text.Text at 0x7f72207bcf90>
```



Numerical solution

first, we need the load function

In [18]:
```
def make_p(p0,t1):
  """make_p(p0,t1) returns a 1/2 sine impulse load function, p(t)"""
  def p(t):
    ""
    if t<t1:
      return p0*sin(t*pi/t1)
    else:
      return 0.0
  return p
```

and also a function that, given the displacement, the velocity and the total plastic deformation, returns the stiffness and the new p.d.; this function is defined in terms of the initial stiffness and the yielding load

In [19]:
```
def make_kt(k,fy):
  "make_kt(k,fy) returns a function kt(u,v,up) returning kt, up"
  def kt(u,v,up):
    f=k*(u-up)
    if (-fy)<f<fy:                return k,up
    if fy<=f    and v>0:  up=u-uy;return 0,up
    if fy<=f    and v<=0: up=u-uy;return k,up
    if f<=(-fy) and v<0:  up=u+uy;return 0,up
    else:                 up=u+uy;return k,up
  return kt
```

Problem data

In [20]:
```
# Exercise from lesson 04
#
mass = 1000.00       # kilograms
k   =  40000.00      # Newtons per metre
zeta  =    0.03      # zeta is the damping ratio
fy =   2500.00       # yelding force, Newtons
t1 =      0.30       # half-sine impulse duration, seconds
p0 =   6000.00       # half-sine impulse peak value, Newtons
uy =      fy/k       # yelding displacement, metres
```

Initialize the algorithm

I. compute the functions that return the load and the tangent sstiffness + plastic deformation
II. compute the damping constant
III. for a given time step, compute all the relevant algorithmic constants, with $\gamma = \frac{1}{2}$ and $\beta = \frac{1}{4}$

In [21]:
```
# using the above constants, define the loading function
p=make_p(p0,t1)
# the following function, given the final displacement, the final
# velocity and the initial plastic deformation returns a) the tangent
# stiffness b) the final plastic deformation
kt=make_kt(k,fy)

# we need the damping coefficient "c", to compute its value from the
# damping ratio we must first compute the undamped natural frequency
wn=sqrt(k/mass)       # natural frequency of the undamped system
damp=2*mass*wn*zeta         # the damping coefficient

# the time step
h=0.005
# required duration for the response
t_end = 4.0
# the number of time steps to arrive at t_end
nsteps=int((t_end+h/100)/h)+1
# the maximum number of iterations in the Newton-Raphson procedure
maxiters = 30
# using the constant acceleration algorithm
# below we define the relevant algorithmic constants
gamma=0.5
beta=1./4.
gb=gamma/beta
a=mass/(beta*h)+damp*gb
b=0.5*mass/beta+h*damp*(0.5*gb-1.0)
```

System state initialization

and a bit more, in species we create two empty vectors to hold the computation results

In [22]:
```
t0=0.0
u0=0.0
up=0.0
v0=0.0
p0=p(t0)
(k0, up)=kt(u0,v0,up)
a0=(p0-damp*v0-k0*(u0-up))/mass

time = []; disp = []
```

Iteration

We iterate over time and, if there is a state change, over the single time step to equilibrate the unbalanced loadings

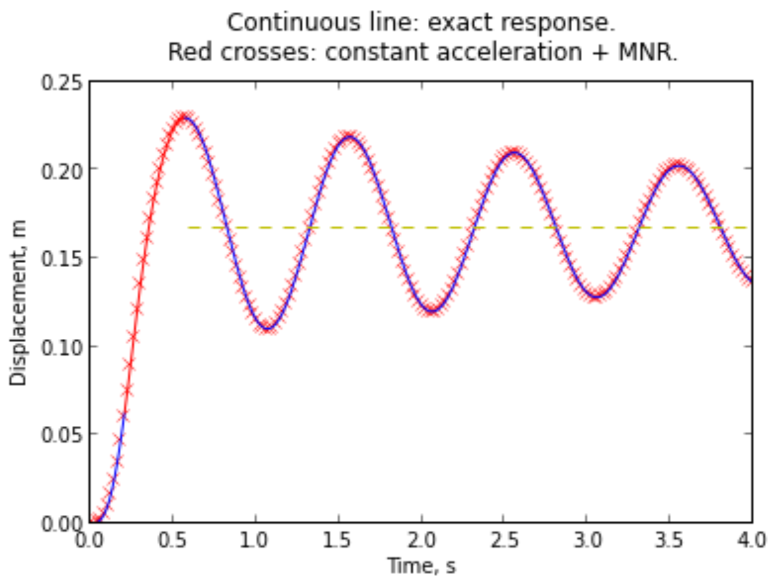
In [23]:
```python
for i in range(nsteps):

    time.append(t0); disp.append(u0)

    # advance time, next external load value, etc
    t1 = t0 + h
    p1 = p(t1)
    Dp = p1 - p0
    Dp_= Dp + a*v0 + b*a0
    k_ = k0 + gb*damp/h + mass/(beta*h*h)
    # we prepare the machinery for the modified Newton-Raphson
    # algorithm.  if we have no state change in the time step, then the
    # N-R algorithm is equivalent to the standard procedure
    u_init=u0; v_init=v0 # initial state
    f_spring=k*(u0-up)   # the force in the spring
    DR=Dp_               # the unbalanced force, initially equal to the
                         # external load increment
    for j in range(maxiters):
        Du=DR/k_          # the disp increment according to the initial stiffne
        u_next = u_init + Du
        v_next = v_init + gb*Du/h - gb*v_init + h*(1.0-0.5*gb)*a0
                          # we are interested in the total plastic elongation
        oops,up=kt(u_next,v_next,up)
                          # because we need the spring force at the end
                          # of the time step
        f_spring_next=k*(u_next-up)
                          # so that we can compute the fraction of the
                          # incremental force that's equilibrated at the
                          # end of the time step
        df=f_spring_next-f_spring+(k_-k0)*Du
                          # and finally the incremental forces unbalanced
                          # at the end of the time step
        DR=DR-df
                          # finish updating the system state
        u_init=u_next; v_init=v_next; f_spring=f_spring_next
                          # if the unbalanced load is small enough (the
                          # criteria used in practical programs are
                          # energy based) exit the loop - note that we
                          # have no plasticization/unloading DR==0 at the
                          # end of the first iteration
        if abs(DR)<fy*1E-6: break
    # now the load increment is balanced by the spring force and
    # increments in inertial and damping forces, we need to compute the
    # full state at the end of the time step, and to change all
    # denominations to reflect the fact that we are starting a new time step.
    Du=u_init-u0
    Dv=gamma*Du/(beta*h)-gamma*v0/beta+h*(1.0-0.5*gamma/beta)*a0
    u1=u0+Du ; v1=v0+Dv
    k1,up=kt(u1,v1,up)
    a1=(p(t1)-damp*v1-k*(u1-up))/mass
    t0=t1; v0=v1; u0=u1 ; a0=a1 ; k0=k1 ; p0=p1
```

Plotting our results

we plot red crosses for the numericaly computed response and a continuous line for the results of the analytical integration of the equation of motion.

```
In [24]: figure(8)
         plot(time[::4],disp[::4],'xr')
         plot(t_el, x_el, '-b',
              t_y1, x_y1, '-r',
              t_y2, x_y2, '-r',
              t_e2, x_e2, '-b',
              (0.6, 4.0), (x0-xy, x0-xy), '--y')
         title("Continuous line: exact response.\n"+
               "Red crosses: constant acceleration + MNR.\n")
         xlabel("Time, s")
         ylabel("Displacement, m")
```

```
Out[24]: <matplotlib.text.Text at 0x3cf1b90>
```


Continuous line: exact response.
Red crosses: constant acceleration + MNR.
Displacement, m
Time, s