Eart119 Homework 5

Sofia Johannesson

May 2019

Differential equation

The differential equation solved in this homework is:

$$y'' + \omega_0^2 y = F cos(\omega t) \tag{1}$$

where F = 0.5, ω_0=0.8 and ω=1.

Part I

Exact solution given in assignment:

$$y(t) = c_1 cos(\omega_0 t) + c_2 sin(\omega_0 t) + \frac{F}{\omega_0^2 - \omega^2} cos(\omega t) \tag{2}$$

The constants c_1 and c_2 are obtained from the initial conditions $y(0) = 0$ and $y'(0) = 0$.

$$c_1 = -\frac{F}{\omega_0^2 - \omega^2} \qquad \& \qquad c_2 = 0 \tag{3}$$

Which means that the general solution will be:

$$y(t) = \frac{F}{\omega_0^2 - \omega^2}(cos(\omega t) - cos(\omega_0 t)) = \frac{F}{\omega_0^2 - \omega^2}(-2) sin(\frac{(\omega + \omega_0)t}{2}) sin(\frac{(\omega - \omega_0)t}{2}) \tag{4}$$

The analytical solution is plotted in figure 1 together with the forcing function (blue) and the slow frequency on its own.

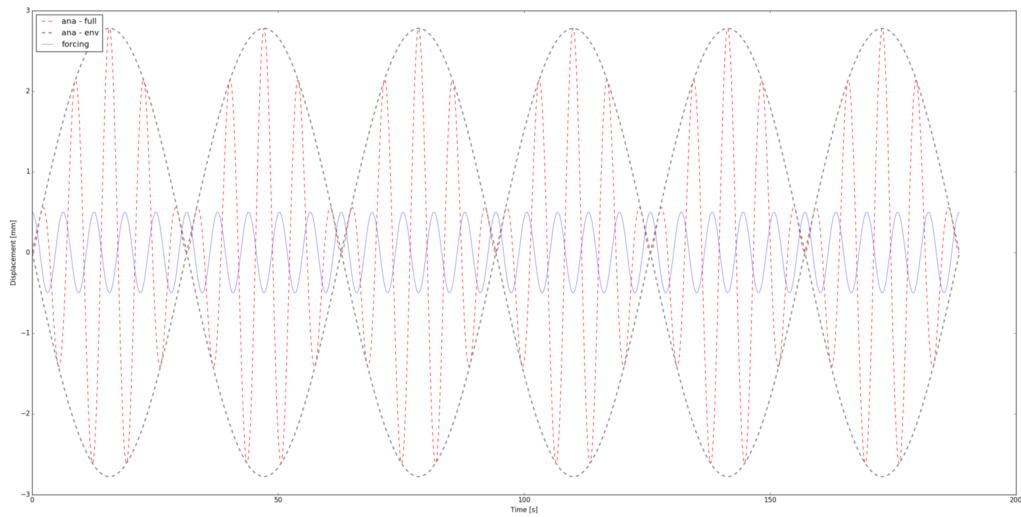

Figure 1: The figure shows the analytical solutions to the differential equation in equation 1

The equation was solved numerically using forward Eulers method and the Runge-Kutta method. Note that when solving a system of ODE's using the Runge-Kutta all k_1 values for all functions evaluated need to be calculated before the k_2 values can be calculated. The numerical solutions can be seen in figure 2.

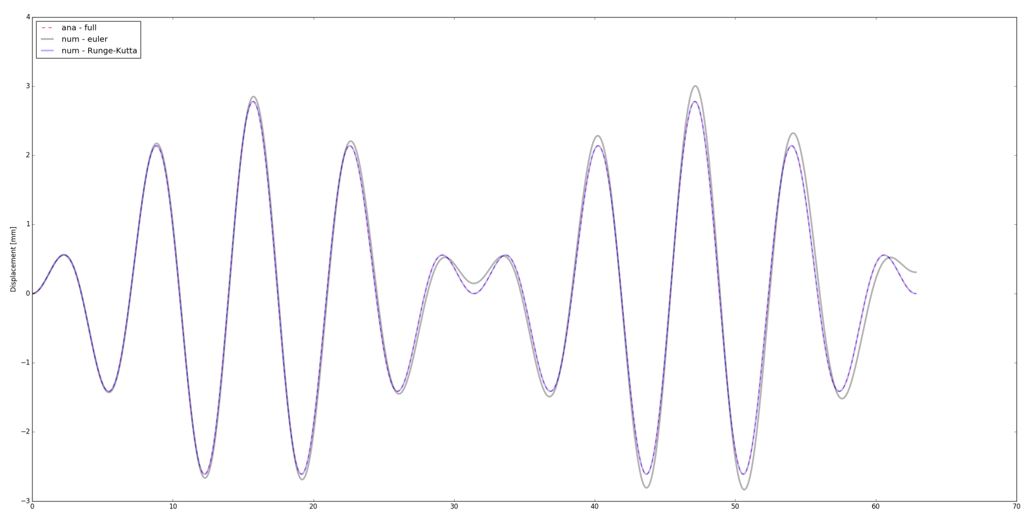

Figure 2: The figure shows the numerical solutions to the above defined differential equation compared to the analytical solution.

In order to see at which timestep the forward Euler became accurate a number of numerical solutions were calculated for different timesteps, h. The result of this can be seen in figure 3.

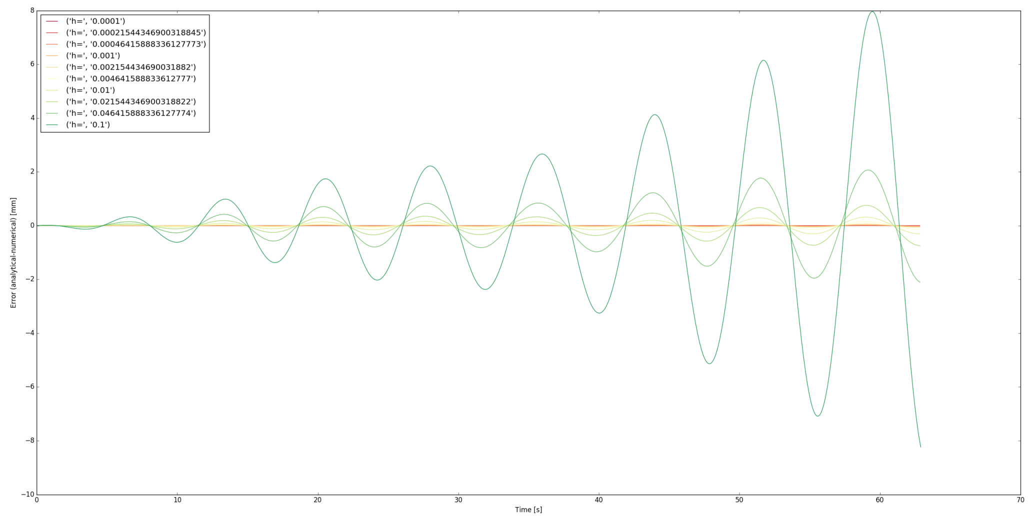

Figure 3: The graphs show the difference between the analytical solution and the nummerical solution using eulers method with different timestep sizes.

All the code for part I can be seen in appendix A.

Part II

When the frequency of the forcing function is in phase with the natural frequency resonance occurs. The analytical solution will in this case, $\omega = \omega_0$, is:

$$y(t) = c_1 cos(\omega_0 t) + c_2 sin(\omega_0 t) + \frac{F}{2\omega_0} t sin(\omega_0 t) \tag{5}$$

Using the initial conditions $y(0) = 0$ and $y'(0) = 0$ gives the values of c_1=0 and c_2=0 which leaves the analytical solution to be:

$$y(t) = \frac{F}{2\omega_0} t sin(\omega_0 t) \tag{6}$$

The result of using $\omega = \omega_0 = 1$ in the numerical calculations can be seen in the figure 4. Eulers method obtains an error that grows larger with time.

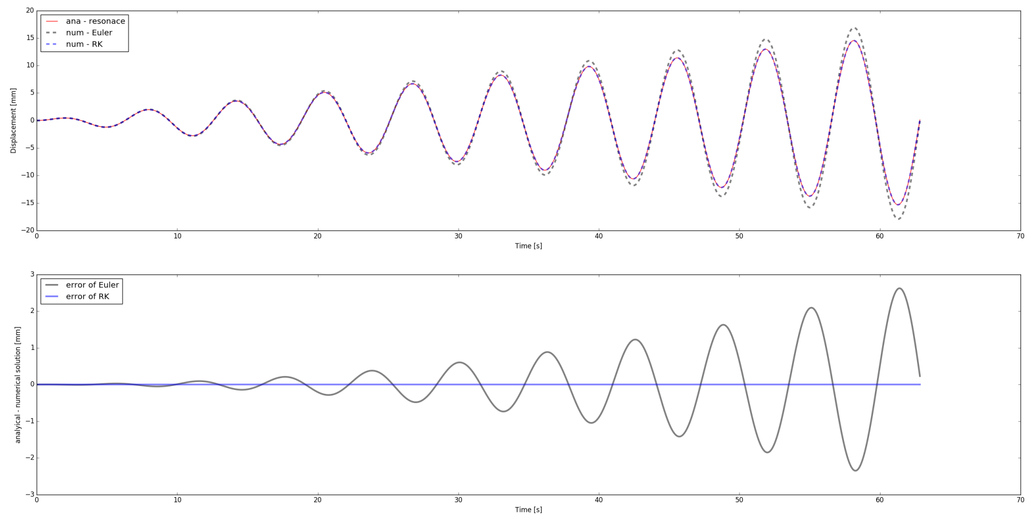

Figure 4: The upper graph shows the analytical solution, red line, and the numerical solutions, dashed lines. The blue graph, corresponding to the Runge-Kutta method follow the behaviour of the analytical solution better than the solution rendered by Euler's method. The lower graph displays the difference between the analytical solution and the numerical solutions. In both graphs, the grey line represents Euler's method and the blue line prepresents the Runge-Kutta method.

Hookes law states that there is linearity between force and change in displacement, which is true for a certain region. But every spring has its limits and after a certain amplitude the spring will stretch or break and the linearity is lost.

The code for this part is in appendix B.

1 Code

A Part I

```
#!/bin/python2.7
"""

solve second order, non-homogeneous ODE: undamped harmonic oscillator

- compare analytical and numerical

ODE: my''(t) + gy'(t) + ky(t) = f(t)
         w0**2 = k/m

ana. solution:
     y(t) = F/(w0**2 - w**2)*( cos(w*t) - cos(w0*t) )
     as a product of sin
     y(t) = F/(w0**2-w**2)*(-2)*sin((w+w0)*t/2) * sin((w-w0)*t/2)

     comparison of Euler and Runge-Kutta numerical solving methods for ODEs
```

```
"""
from __future__ import division

import os
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

#---------------------------------0---------------------------------------
#                          fct defintion
#-------------------------------------------------------------------------
#import ODE.ode_utils as utils

#:TODO - create a python method that solves the ODE system using Runge-Kutta Method

def num_sol( at, y0, par):
    """
    - solve second order ODE for forced, undamped oscillation by solving two first order (
       ODE:  y''(t) + ky(t) = f(t)
                                f(t) = F*cos(w*t)
    :param y0:          - IC
    :param at  :        - time vector
    :param par :        - dictionary with fct parameters
    :return: ay_hat     - forward Euler
    """
    nSteps    = at.shape[0]
    # create vectors for displacement and velocity
    au_hat    = np.zeros( nSteps) #displacement
    av_hat    = np.zeros( at.shape[0]) #velocity
    #initial conditions
    au_hat[0] = y0[0]
    av_hat[0] = y0[1]
    for i in range( nSteps-1):
        #slope at previous time step, i. RHS of ODE system
        fn1 = av_hat[i]
        fn2 = dPar['F']*np.cos(dPar['w']*at[i])-dPar['w0']**2*au_hat[i]
        # Euler formula: y[n+1] = y[n] + fn*h
        au_hat[i+1] = au_hat[i]+fn1*dPar['h']
        av_hat[i+1] = av_hat[i]+fn2*dPar['h']
        #alternatively use Euler-Cromer: see Langtangen & Linge, p 129, eq: 4.49 - 4.52
        #av_hat[i+1] = av_hat[i] + fn2*par['h']
        #au_hat[i+1] = au_hat[i] + av_hat[i+1]*par['h']
    return au_hat, av_hat

def num_sol_RK( at, y0, par):
    """
    - solve second order ODE for forced, undamped oscillation by solving two first order (
       ODE:  y''(t) + ky(t) = f(t)
                                f(t) = F*cos(w*t)
    :param y0:          - IC
    :param at  :        - time vector
    :param par :        - dictionary with fct parameters
```

```
    :return: ay_hat      - Runge-Kutta
    """
    nSteps      = at.shape[0]
    # create vectors for displacement and velocity
    au_hat      = np.zeros( nSteps) #displacement
    av_hat      = np.zeros( at.shape[0]) #velocity

    def dudt(t,v):
        return v

    def dvdt(t,u):
        return dPar['F']*np.cos(dPar['w']*t)-dPar['w0']**2*u
    #initial conditions
    au_hat[0] = y0[0]
    av_hat[0] = y0[1]
    for i in range( nSteps-1):
        #slope at previous time step, i. RHS of ODE system
        #have to evaluate u,v simultaniously
        kn1u = dudt(at[i], av_hat[i])
        kn1v = dvdt(at[i], au_hat[i])
        kn2u = dudt(at[i]+0.5*dPar['h'],av_hat[i]+0.5*dPar['h']*kn1v)
        kn2v = dvdt(at[i]+0.5*dPar['h'],au_hat[i]+0.5*dPar['h']*kn1u)
        kn3u = dudt(at[i]+0.5*dPar['h'],av_hat[i]+0.5*dPar['h']*kn2v)
        kn3v = dvdt(at[i]+0.5*dPar['h'],au_hat[i]+0.5*dPar['h']*kn2u)
        kn4u = dudt(at[i]+dPar['h'],av_hat[i]+dPar['h']*kn3v)
        kn4v = dvdt(at[i]+dPar['h'],au_hat[i]+dPar['h']*kn3u)


        # Runge-Kutta
        au_hat[i+1] = au_hat[i]+( (kn1u + 2*kn2u + 2*kn3u + kn4u)/6)*dPar['h']
        av_hat[i+1] = av_hat[i]+( (kn1v + 2*kn2v + 2*kn3v + kn4v)/6)*dPar['h']
        #alternatively use Euler-Cromer: see Langtangen & Linge, p 129, eq: 4.49 - 4.52
        #av_hat[i+1] = av_hat[i] + fn2*par['h']
        #au_hat[i+1] = au_hat[i] + av_hat[i+1]*par['h']
    return au_hat, av_hat
#------------------------------------1------------------------------------------
#                         params, files, dir
#-------------------------------------------------------------------------------
dPar = {     #frequencies
             'w0' :    .8, #= sqrt(k/m) --> natural frequency
             'w'  :    1, # --> frequency of forcing function
             # forcing function amplitude
             'F'  : 0.5, #20,
             # initial conditions for displ. and velocity
             'y01' : 0, 'y02' : 0,
             # time stepping
             'h'  : 1e-2,
             'tStart' : 0,
             'tStop'  : 20*np.pi,}

#------------------------------------2------------------------------------------
#                           analytical solution
#-------------------------------------------------------------------------------
```

```
at = np.arange( dPar['tStart'], dPar['tStop']+dPar['h'], dPar['h'])
# forcing function
fct_t     = dPar['F']*np.cos( dPar['w']*at)
#the analytical solution
preFacAna = dPar['F']/(dPar['w0']**2 - dPar['w']**2)
ay_ana1   = preFacAna * ((-2)*np.sin((dPar['w']+dPar['w0'])*at/2)*np.sin((dPar['w']-dPar['

# envelope - slow frequency
ay_ana2 = -2*preFacAna* ( np.sin( (dPar['w']-dPar['w0'])/2*at))

#----------------------------------3----------------------------------------
#                        numerical solutions
#--------------------------------------------------------------------------
#euler
aU_num, aV_num = num_sol(   at, [dPar['y01'], dPar['y02']], dPar)

#RK
aU_numRK, aV_numRK = num_sol_RK(   at, [dPar['y01'], dPar['y02']], dPar)



#----------------------------------4----------------------------------------
#                              plots
#--------------------------------------------------------------------------
plt.figure(1) #just analytical solutions
ax = plt.subplot( 111) #plt.axes( [.12, .12, .83, .83])
#ax.plot( at, aU_num,    'k-', lw = 3, alpha = .3, label = 'num - displ.')
ax.plot( at,  ay_ana1, 'r--', lw = 1, label = 'ana - full')
ax.plot( at,  ay_ana2,  '--', color = '.5', lw = 2, label = 'ana - env')
ax.plot( at,  -ay_ana2, '--', color = '.5', lw = 2)
ax.plot( at, fct_t, 'b-', lw = 1, alpha = .5, label ='forcing')
ax.set_xlabel( 'Time [s]')
ax.set_ylabel( 'Displacement [mm]')
ax.legend( loc = 'upper left')
plt.show()

plt.figure(2) #numerical solutions
ax2 = plt.subplot( 111)
ax2.plot( at,  ay_ana1, 'r--', lw = 1, label = 'ana - full')
ax2.plot( at, aU_num,    'k-', lw = 3, alpha = .3, label = 'num - euler')
ax2.plot(at, aU_numRK, 'b-', lw=3,alpha =0.3, label='num - Runge-Kutta')
ax2.set_xlabel( 'Time [s]')
ax2.set_ylabel( 'Displacement [mm]')
ax2.legend( loc = 'upper left')
plt.show()

plt.figure(3)
ax3 = plt.subplot(111)
ax3.set_xlabel( 'Time [s]')
ax3.set_ylabel( 'Error (analytical-numerical) [mm]')

# continuous color scale
kspace = np.linspace(-4,-1,10)
```

```
cmin,cmax      = 0, len( kspace)
cmap           = plt.cm.RdYlGn
normCmap       = mpl.colors.Normalize( vmin=cmin, vmax=cmax )
scalarMap      = mpl.cm.ScalarMappable(norm=normCmap, cmap=plt.get_cmap(cmap) )
aC = scalarMap.to_rgba( np.arange( (cmax-cmin)) )

#vary timestep for euler
i = -1
for k in kspace:
    i += 1
    h = 10**(k)
    print h
    dPar['h'] = h
    at = np.arange( dPar['tStart'], dPar['tStop']+dPar['h'], dPar['h'])
    aU_num, aV_num = num_sol(at, [dPar['y01'], dPar['y02']], dPar)
    ay_ana= preFacAna * ((-2)*np.sin((dPar['w']+dPar['w0'])*at/2)*np.sin((dPar['w']-dPar['
    err = ay_ana - aU_num
    grafname = ('h=', str(h))
    ax3.plot( at,  err, color = aC[i], lw = 1, label = grafname)
    #ax3.plot( at,  err)
ax3.legend( loc = 'upper left')
plt.show()
```

B Part II

```
# -*- coding: utf-8 -*-
"""

solve second order, non-homogeneous ODE: undamped harmonic oscillator

- compare analytical and numerical for resonance

ODE: my''(t) + gy'(t) + ky(t) = f(t)
       w0**2 = k/m

ana. solution (at resonance)
    y(t) = F/(2w0)*t*sin(w0*t)

    comparison of Euler and Runge-Kutta numerical solving methods for ODEs

"""
from __future__ import division

import os
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

#-----------------------------------0-------------------------------------------
#                          fct defintion
#-------------------------------------------------------------------------------
#import ODE.ode_utils as utils
```

```
#:TODO - create a python method that solves the ODE system using Runge-Kutta Method

def num_sol( at, y0, par):
    """
    - solve second order ODE for forced, undamped oscillation by solving two first order O
       ODE:  y''(t) + ky(t) = f(t)
                              f(t) = F*cos(w*t)
    :param y0:          - IC
    :param at   :       - time vector
    :param par :        - dictionary with fct parameters
    :return: ay_hat     - forward Euler
    """
    nSteps    = at.shape[0]
    # create vectors for displacement and velocity
    au_hat    = np.zeros( nSteps) #displacement
    av_hat    = np.zeros( at.shape[0]) #velocity
    #initial conditions
    au_hat[0] = y0[0]
    av_hat[0] = y0[1]
    for i in range( nSteps-1):
        #slope at previous time step, i. RHS of ODE system
        fn1 = av_hat[i]
        fn2 = dPar['F']*np.cos(dPar['w']*at[i])-dPar['w0']**2*au_hat[i]
        # Euler formula: y[n+1] = y[n] + fn*h
        au_hat[i+1] = au_hat[i]+fn1*dPar['h']
        av_hat[i+1] = av_hat[i]+fn2*dPar['h']
        #alternatively use Euler-Cromer: see Langtangen & Linge, p 129, eq: 4.49 - 4.52
        #av_hat[i+1] = av_hat[i] + fn2*par['h']
        #au_hat[i+1] = au_hat[i] + av_hat[i+1]*par['h']
    return au_hat, av_hat

def num_sol_RK( at, y0, par):
    """
    - solve second order ODE for forced, undamped oscillation by solving two first order O
       ODE:  y''(t) + ky(t) = f(t)
                              f(t) = F*cos(w*t)
    :param y0:          - IC
    :param at   :       - time vector
    :param par :        - dictionary with fct parameters
    :return: ay_hat     - Runge-Kutta
    """
    nSteps    = at.shape[0]
    # create vectors for displacement and velocity
    au_hat    = np.zeros( nSteps) #displacement
    av_hat    = np.zeros( at.shape[0]) #velocity

    def dudt(t,v):
        return v

    def dvdt(t,u):
        return dPar['F']*np.cos(dPar['w']*t)-dPar['w0']**2*u
    #initial conditions
    au_hat[0] = y0[0]
```

```python
    av_hat[0] = y0[1]
    for i in range( nSteps-1):
        #slope at previous time step, i. RHS of ODE system
        #have to evaluate u,v simultaniously
        kn1u = dudt(at[i], av_hat[i])
        kn1v = dvdt(at[i], au_hat[i])
        kn2u = dudt(at[i]+0.5*dPar['h'],av_hat[i]+0.5*dPar['h']*kn1v)
        kn2v = dvdt(at[i]+0.5*dPar['h'],au_hat[i]+0.5*dPar['h']*kn1u)
        kn3u = dudt(at[i]+0.5*dPar['h'],av_hat[i]+0.5*dPar['h']*kn2v)
        kn3v = dvdt(at[i]+0.5*dPar['h'],au_hat[i]+0.5*dPar['h']*kn2u)
        kn4u = dudt(at[i]+dPar['h'],av_hat[i]+dPar['h']*kn3v)
        kn4v = dvdt(at[i]+dPar['h'],au_hat[i]+dPar['h']*kn3u)


        # Runge-Kutta
        au_hat[i+1] = au_hat[i]+( (kn1u + 2*kn2u + 2*kn3u + kn4u)/6)*dPar['h']
        av_hat[i+1] = av_hat[i]+( (kn1v + 2*kn2v + 2*kn3v + kn4v)/6)*dPar['h']
        #alternatively use Euler-Cromer: see Langtangen & Linge, p 129, eq: 4.49 - 4.52
        #av_hat[i+1] = av_hat[i] + fn2*par['h']
        #au_hat[i+1] = au_hat[i] + av_hat[i+1]*par['h']
    return au_hat, av_hat
#--------------------------------------1-------------------------------------------
#                       params, files, dir
#----------------------------------------------------------------------------------
dPar = {    #frequencies
            'w0' :   1, #= sqrt(k/m) --> natural frequency
            'w'  :   1, # --> frequency of forcing function
            # forcing function amplitude
            'F'  : 0.5, #20,
            # initial conditions for displ. and velocity
            'y01' : 0, 'y02' : 0,
            # time stepping
            'h'  : 1e-2,
            'tStart' : 0,
            'tStop'  : 20*np.pi,}

#--------------------------------------2-------------------------------------------
#                           analytical solution
#----------------------------------------------------------------------------------
at = np.arange( dPar['tStart'], dPar['tStop']+dPar['h'], dPar['h'])
# forcing function
fct_t      = dPar['F']*np.cos( dPar['w']*at)
#the analytical solution
ay_anaRes   = dPar['F']/(2*dPar['w0'])*at*np.sin(dPar['w0']*at)

#--------------------------------------3-------------------------------------------
#                           numerical solutions
#----------------------------------------------------------------------------------
#euler
aU_num, aV_num = num_sol(  at, [dPar['y01'], dPar['y02']], dPar)

#RK
aU_numRK, aV_numRK = num_sol_RK(  at, [dPar['y01'], dPar['y02']], dPar)
```

```
#-----------------------------------4-----------------------------------------
#                                 plots
#-----------------------------------------------------------------------------
plt.figure(1) #just analytical solutions
ax = plt.subplot( 211) #plt.axes( [.12, .12, .83, .83])
#ax.plot( at, aU_num,    'k-', lw = 3, alpha = .3, label = 'num - displ.')
ax.plot( at,  ay_anaRes, 'r', lw = 1, label = 'ana - resonace')
ax.plot(at, aU_num, 'k--', lw=3, alpha = 0.5, label= 'num - Euler')
ax.plot(at, aU_numRK, 'b--', lw=3, alpha=0.5, label = 'num - RK')
ax.set_xlabel( 'Time [s]')
ax.set_ylabel( 'Displacement [mm]')
ax.legend( loc = 'upper left')
ax2 = plt.subplot(212) #difference between nummerical and analytical solution
ax2.plot(at, ay_anaRes-aU_num, 'k', lw=3, alpha = 0.5, label= 'error of Euler')
ax2.plot(at, ay_anaRes-aU_numRK, 'b', lw=3, alpha = 0.5, label= 'error of RK')
ax2.set_xlabel( 'Time [s]')
ax2.set_ylabel( 'analyical - numerical solution [mm]')
ax2.legend( loc = 'upper left')
plt.show()
```