

# Study Guide: Vibration ODEs

Hans Petter Langtangen<sup>1,2</sup>

Center for Biomedical Computing, Simula Research Laboratory<sup>1</sup>

Department of Informatics, University of Oslo<sup>2</sup>

Sep 4, 2013

## A simple vibration problem

$$u'' + \omega^2 u = 0, \quad u(0) = I, \quad u'(0) = 0, \quad t \in (0, T]. \quad (1)$$

Exact solution:

$$u(t) = I \cos(\omega t). \quad (2)$$

$u(t)$  oscillates with constant amplitude  $I$  and (angular) frequency  $\omega$ . Period:  $P = 2\pi/\omega$ .

## A centered finite difference scheme; step 1 and 2

- Strategy: follow the four steps of the finite difference method.
- Step 1: Introduce a time mesh, here uniform on  $[0, T]$ :  
 $t_n = n\Delta t$
- Step 2: Let the ODE be satisfied at each mesh point:

$$u''(t_n) + \omega^2 u(t_n) = 0, \quad n = 1, \dots, N_t. \quad (3)$$

## A centered finite difference scheme; step 3

Step 3: Approximate derivative(s) by finite difference approximation(s). Very common (standard!) formula for  $u''$ :

$$u''(t_n) \approx \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2}. \quad (4)$$

Use this discrete initial condition together with the ODE at  $t = 0$  to eliminate  $u^{-1}$  (insert (??) in (??)):

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = -\omega^2 u^n. \quad (5)$$

## A centered finite difference scheme; step 4

Step 4: Formulate the computational algorithm. Assume  $u^{n-1}$  and  $u^n$  are known, solve for unknown  $u^{n+1}$ :

$$u^{n+1} = 2u^n - u^{n-1} - \omega^2 u^n. \quad (6)$$

Nick names for this scheme: Stormer's method or Verlet integration.

# Computing the first step

- The formula breaks down for  $u^1$  because  $u^{-1}$  is unknown and outside the mesh!
- And: we have not used the initial condition  $u'(0) = 0$ .

Discretize  $u'(0)$  by a centered difference

$$\frac{u^1 - u^{-1}}{2\Delta t} = 0 \quad \Rightarrow \quad u^{-1} = u^1. \quad (7)$$

Inserted in (??) for  $n = 0$  gives

$$u^1 = u^0 - \frac{1}{2}\Delta t^2 \omega^2 u^0. \quad (8)$$

# The computational algorithm

- 1  $u^0 = I$
- 2 compute  $u^1$  from (??)
- 3 for  $n = 1, 2, \dots, N_t - 1$ :
  - 1 compute  $u^{n+1}$  from (??)

More precisely expressed in Python:

```
t = linspace(0, T, Nt+1) # mesh points in time
dt = t[1] - t[0]         # constant time step.
u = zeros(Nt+1)          # solution

u[0] = I
u[1] = u[0] - 0.5*dt**2*w**2*u[0]
for n in range(1, Nt):
    u[n+1] = 2*u[n] - u[n-1] - dt**2*w**2*u[n]
```

Note:  $w$  is consistently used for  $\omega$  in code.

With  $[D_t D_t u]^n$  as the finite difference approximation to  $u''(t_n)$  we can write

$$[D_t D_t u + \omega^2 u = 0]^n. \quad (9)$$

$[D_t D_t u]^n$  means applying a central difference with step  $\Delta t/2$  twice:

$$[D_t(D_t u)]^n = \frac{[D_t u]^{n+1/2} - [D_t u]^{n-1/2}}{\Delta t}$$

which is written out as

$$\frac{1}{\Delta t} \left( \frac{u^{n+1} - u^n}{\Delta t} - \frac{u^n - u^{n-1}}{\Delta t} \right) = \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2}.$$



$$[u = I]^0, \quad [D_{2t}u = 0]^0, \quad (10)$$

where  $[D_{2t}u]^n$  is defined as

$$[D_{2t}u]^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t}. \quad (11)$$

$u$  is often displacement/position,  $u'$  is velocity and can be computed by

$$u'(t_n) \approx \frac{u^{n+1} - u^{n-1}}{2\Delta t} = [D_{2t}u]^n. \quad (12)$$



# Core algorithm

```
from numpy import *
from matplotlib.pyplot import *

def solver(I, w, dt, T):
    """
    Solve  $u'' + w^2 u = 0$  for  $t$  in  $(0, T]$ ,  $u(0)=I$  and  $u'(0)=0$ ,
    by a central finite difference method with time step  $dt$ .
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = zeros(Nt+1)
    t = linspace(0, Nt*dt, Nt+1)

    u[0] = I
    u[1] = u[0] - 0.5*dt**2*w**2*u[0]
    for n in range(1, Nt):
        u[n+1] = 2*u[n] - u[n-1] - dt**2*w**2*u[n]
    return u, t
```

# Plotting

```
def exact_solution(t, I, w):  
    return I*cos(w*t)  
  
def visualize(u, t, I, w):  
    plot(t, u, 'r--o')  
    t_fine = linspace(0, t[-1], 1001) # very fine mesh for u_e  
    u_e = exact_solution(t_fine, I, w)  
    hold('on')  
    plot(t_fine, u_e, 'b-')  
    legend(['numerical', 'exact'], loc='upper left')  
    xlabel('t')  
    ylabel('u')  
    dt = t[1] - t[0]  
    title('dt=%g' % dt)  
    umin = 1.2*u.min(); umax = -umin  
    axis([t[0], t[-1], umin, umax])  
    savefig('vib1.png')  
    savefig('vib1.pdf')  
    savefig('vib1.eps')
```

# Main program

```
I = 1
w = 2*pi
dt = 0.05
num_periods = 5
P = 2*pi/w      # one period
T = P*num_periods
u, t = solver(I, w, dt, T)
visualize(u, t, I, w, dt)
```

## User interface: command line

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--I', type=float, default=1.0)
parser.add_argument('--w', type=float, default=2*pi)
parser.add_argument('--dt', type=float, default=0.05)
parser.add_argument('--num_periods', type=int, default=5)
a = parser.parse_args()
I, w, dt, num_periods = a.I, a.w, a.dt, a.num_periods
```

# Running the program

vib\_undamped.py:

```
Terminal> python vib_undamped.py --dt 0.05 --num_periods 40
```

Generates frames tmp\_vib%04d.png in files. Can make movie:

```
Terminal> avconv -r 12 -i tmp_vib%04d.png -vcodec flv movie.flv
```

Can use ffmpeg instead of avconv.

| <i>Format</i> | <i>Codec and filename</i>          |
|---------------|------------------------------------|
| <i>Flash</i>  | <i>-vcodec flv movie.flv</i>       |
| <i>MP4</i>    | <i>-vcodec libx64 movie.mp4</i>    |
| <i>Webm</i>   | <i>-vcodec libvpx movie.webm</i>   |
| <i>Ogg</i>    | <i>-vcodec libtheora movie.ogg</i> |





# First steps for testing and debugging

- **Testing very simple solutions:**  $u = \text{const}$  or  $u = ct + d$  do not apply here (without a force term in the equation:  $u'' + \omega^2 u = f$ ).
- **Hand calculations:** calculate  $u^1$  and  $u^2$  and compare with program.

# Checking convergence rates

The function below

- performs  $m$  simulations with halved time steps:  $2^{-k}\Delta t$ ,  $k = 0, \dots, m - 1$ ,
- computes the  $L_2$  norm of the error,  
$$E = \sqrt{\Delta t_i \sum_{n=0}^{N_t-1} (u^n - u_e(t_n))^2}$$
 in each case,
- estimates the rates  $r_i$  from two consecutive experiments  $(\Delta t_{i-1}, E_{i-1})$  and  $(\Delta t_i, E_i)$ , assuming  $E_i = C\Delta t_i^{r_i}$  and  $E_{i-1} = C\Delta t_{i-1}^{r_i}$ :

# Implementational details

```
def convergence_rates(m, num_periods=8):  
    """  
    Return m-1 empirical estimates of the convergence rate  
    based on m simulations, where the time step is halved  
    for each simulation.  
    """  
    w = 0.35; I = 0.3  
    dt = 2*pi/w/30 # 30 time step per period 2*pi/w  
    T = 2*pi/w*num_periods  
    dt_values = []  
    E_values = []  
    for i in range(m):  
        u, t = solver(I, w, dt, T)  
        u_e = exact_solution(t, I, w)  
        E = sqrt(dt*sum((u_e-u)**2))  
        dt_values.append(dt)  
        E_values.append(E)  
        dt = dt/2  
  
    r = [log(E_values[i-1]/E_values[i])/  
        log(dt_values[i-1]/dt_values[i])  
        for i in range(1, m, 1)]  
    return r
```

Result: r contains values equal to 2.00 - as expected!

# Nose test

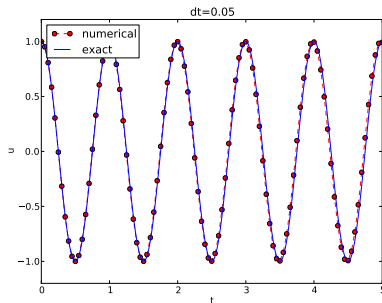
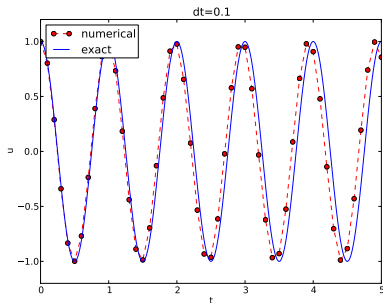
Use final `r[-1]` in a unit test:

```
def test_convergence_rates():  
    r = convergence_rates(m=5, num_periods=8)  
    # Accept rate to 1 decimal place  
    nt.assert_almost_equal(r[-1], 2.0, places=1)
```

Complete code in `vib_undamped.py`.

# Long time simulations

# Effect of the time step on long simulations



- The numerical solution seems to have right amplitude.
- There is a phase error which is reduced by reducing the time step.
- The total phase error seems to grow with time.

# Using a moving plot window

- In long time simulations we need a plot window that follows the solution.
- Method 1: `scitools.MovingPlotWindow`.
- Method 2: `scitools.avplotter` (ASCII vertical plotter).

Example:

```
Terminal> python vib_undamped.py --dt 0.05 --num_periods 40
```

Movie of the moving plot window.



# Analysis of the numerical scheme

## Deriving an exact numerical solution; ideas

- Linear, homogeneous, difference equation for  $u^n$ .
- Has solutions  $u^n \sim A^n$ , where  $A$  is unknown (number).
- Here:  $u_e(t) = I \cos(\omega t) \sim I \exp(i\omega t) = I(e^{i\omega\Delta t})^n$
- Trick for simplifying the algebra:  $u^n = A^n$ , seek  $A = \exp(i\tilde{\omega}\Delta t)$  and find  $\tilde{\omega}$
- $\tilde{\omega}$ : unknown *numerical frequency* (easier to calculate than  $A$ )
- $\omega - \tilde{\omega}$  is the *phase error*
- Use the real part as the physical relevant part of a complex expression

## Deriving an exact numerical solution; calculations (1)

$$u^n = A^n = \exp(\tilde{\omega} \Delta t n) = \exp(\tilde{\omega} t) = \cos(\tilde{\omega} t) + i \sin(\tilde{\omega} t).$$

$$\begin{aligned} [D_t D_t u]^n &= \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} \\ &= I \frac{A^{n+1} - 2A^n + A^{n-1}}{\Delta t^2} \\ &= I \frac{\exp(i\tilde{\omega}(t + \Delta t)) - 2\exp(i\tilde{\omega}t) + \exp(i\tilde{\omega}(t - \Delta t))}{\Delta t^2} \\ &= I \exp(i\tilde{\omega}t) \frac{1}{\Delta t^2} (\exp(i\tilde{\omega}(\Delta t)) + \exp(i\tilde{\omega}(-\Delta t)) - 2) \\ &= I \exp(i\tilde{\omega}t) \frac{2}{\Delta t^2} (\cosh(i\tilde{\omega}\Delta t) - 1) \\ &= I \exp(i\tilde{\omega}t) \frac{2}{\Delta t^2} (\cos(\tilde{\omega}\Delta t) - 1) \\ &= -I \exp(i\tilde{\omega}t) \frac{4}{\Delta t^2} \sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) \end{aligned}$$

## Deriving an exact numerical; calculations (2)

The scheme (??) with  $u^n = \exp(i\omega\tilde{\Delta}t n)$  inserted gives

$$-\exp(i\tilde{\omega}t) \frac{4}{\Delta t^2} \sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) + \omega^2 \exp(i\tilde{\omega}t) = 0, \quad (13)$$

which after dividing by  $\exp(i\tilde{\omega}t)$  results in

$$\frac{4}{\Delta t^2} \sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = \omega^2. \quad (14)$$

Solve for  $\tilde{\omega}$ :

$$\tilde{\omega} = \pm \frac{2}{\Delta t} \sin^{-1}\left(\frac{\omega\Delta t}{2}\right). \quad (15)$$

- Phase error because  $\tilde{\omega} \neq \omega$ .
- But how good is the approximation  $\tilde{\omega}$  to  $\omega$ ?

# Polynomial approximation of the phase error

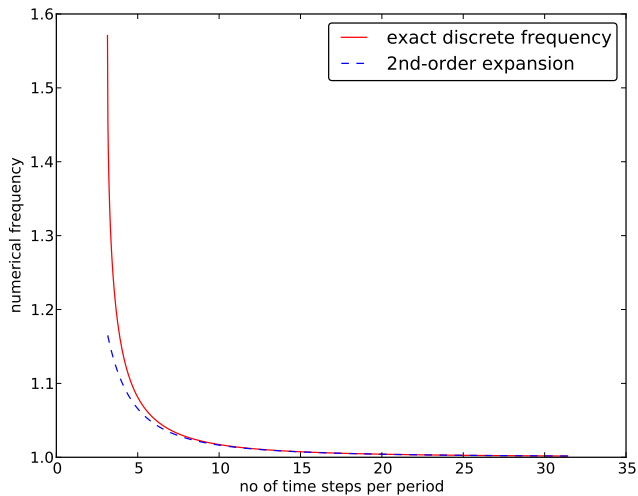
Taylor series expansion for small  $\Delta t$  gives a formula that is easier to understand:

```
>>> from sympy import *
>>> dt, w = symbols('dt w')
>>> w_tilde = asin(w*dt/2).series(dt, 0, 4)*2/dt
>>> print w_tilde
(dt*w + dt**3*w**3/24 + O(dt**4))/dt # observe final /dt
```

$$\tilde{\omega} = \omega \left( 1 + \frac{1}{24} \omega^2 \Delta t^2 \right) + \mathcal{O}(\Delta t^3). \quad (16)$$

The numerical frequency is too large (to fast oscillations).

# Plot of the phase error



$$u^n = I \cos(\tilde{\omega} n \Delta t), \quad \tilde{\omega} = \frac{2}{\Delta t} \sin^{-1} \left( \frac{\omega \Delta t}{2} \right). \quad (17)$$

Ideal for verification and analysis!

- Numerical solution has constant amplitude (desired!), but phase error.
- Constant amplitude requires  $\sin^{-1}(\omega\Delta t/2)$  to be real-valued  $\Rightarrow |\omega\Delta t/2| \leq 1$ .
- $\sin^{-1}(x)$  is complex if  $|x| > 1$ , and then  $\tilde{\omega}$  becomes complex.

What is the consequence of complex  $\tilde{\omega}$ ?

- Set  $\tilde{\omega} = \tilde{\omega}_r + i\tilde{\omega}_i$ .
- Since  $\sin^{-1}(x)$  has a negative\* imaginary part for  $x > 1$ ,  $\exp(i\omega\tilde{t}) = \exp(-\tilde{\omega}_i t) \exp(i\tilde{\omega}_r t)$  leads to exponential growth  $e^{-\tilde{\omega}_i t}$  when  $-\tilde{\omega}_i t > 0$ .
- This instability because the qualitative behavior is wrong.

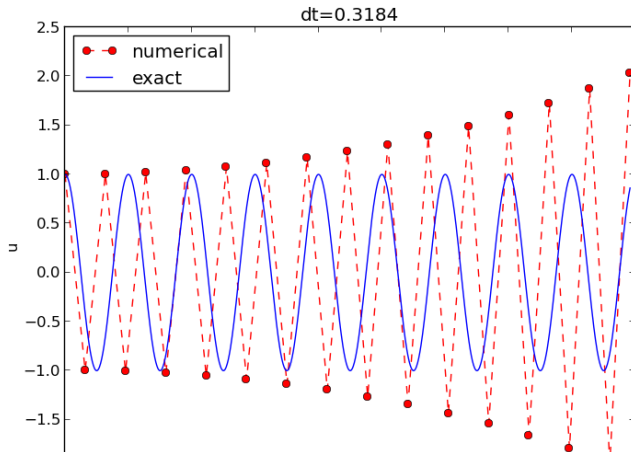


# The stability criterion

Cannot tolerate growth and must therefore demand a *stability criterion*

$$\frac{\omega \Delta t}{2} \leq 1 \quad \Rightarrow \quad \Delta t \leq \frac{2}{\omega}. \quad (18)$$

Try  $\Delta t = \frac{2}{\omega} + 9.01 \cdot 10^{-5} (\pi^{-1})$ :



# Summary of the analysis

We can draw three important conclusions:

- ❶ The key parameter in the formulas is  $p = \omega \Delta t$ .
  - ❶ Period of oscillations:  $P = 2\pi/\omega$
  - ❷ Number of time steps per period:  $N_P = P/\Delta t$
  - ❸  $\Rightarrow p = \omega \Delta t = 2\pi N_P \sim N_P$
  - ❹ The smallest possible  $N_P$  is 2  $\Rightarrow p \in (0, \pi]$
- ❷ For  $p \leq 2$  the amplitude of  $u^n$  is constant (stable solution)
- ❸  $u^n$  has a relative phase error  $\tilde{\omega}/\omega \approx 1 + \frac{1}{24}p^2$ , making numerical peaks occur too early

# Alternative schemes based on 1st-order equations

## Rewriting 2nd-order ODE as system of two 1st-order ODEs

$$u'' + \omega^2 u = 0$$

Introduce auxiliary variable  $v = u'$ :

$$u' = v, \tag{19}$$

$$v' = -\omega^2 u. \tag{20}$$

Initial conditions:  $u(0) = I$  and  $v(0) = 0$ .

# The Forward Euler scheme

$$\begin{aligned} [D_t^+ u &= v]^n, \\ [D_t^+ v &= -\omega^2 u]^n, \end{aligned}$$

or written out,

$$u^{n+1} = u^n + \Delta t v^n, \tag{21}$$

$$v^{n+1} = v^n - \Delta t \omega^2 u^n. \tag{22}$$

# The Backward Euler scheme

$$[D_t^- u = v]^{n+1}, \quad (23)$$

$$[D_t^- v = -\omega u]^{n+1}. \quad (24)$$

Written out:

$$u^{n+1} - \Delta t v^{n+1} = u^n, \quad (25)$$

$$v^{n+1} + \Delta t \omega^2 u^{n+1} = v^n. \quad (26)$$

This is a *coupled*  $2 \times 2$  system for the new values at  $t = t_{n+1}$ !

# The Crank-Nicolson scheme

$$[D_t u = \bar{v}^t]^{n+\frac{1}{2}}, \quad (27)$$

$$[D_t v = -\omega \bar{u}^t]^{n+\frac{1}{2}}. \quad (28)$$

Also a coupled system:

$$u^{n+1} - \frac{1}{2}\Delta t v^{n+1} = u^n + \frac{1}{2}\Delta t v^n, \quad (29)$$

$$v^{n+1} + \frac{1}{2}\Delta t \omega^2 u^{n+1} = v^n - \frac{1}{2}\Delta t \omega^2 u^n. \quad (30)$$

# Comparison of schemes via Odespy

Can use Odespy to compare many methods for first-order schemes:

```
import odespy
import numpy as np

def f(u, t, w=1):
    # u is array of length 2 holding our [u, v]
    u, v = u
    return [v, -w**2*u]

def run_solvers_and_plot(solvers, timesteps_per_period=20,
                        num_periods=1, I=1, w=2*np.pi):
    P = 2*np.pi/w # one period
    dt = P/timesteps_per_period
    Nt = num_periods*timesteps_per_period
    T = Nt*dt
    t_mesh = np.linspace(0, T, Nt+1)

    legends = []
    for solver in solvers:
        solver.set(f_kwargs={'w': w})
        solver.set_initial_condition([I, 0])
        u, t = solver.solve(t_mesh)
```



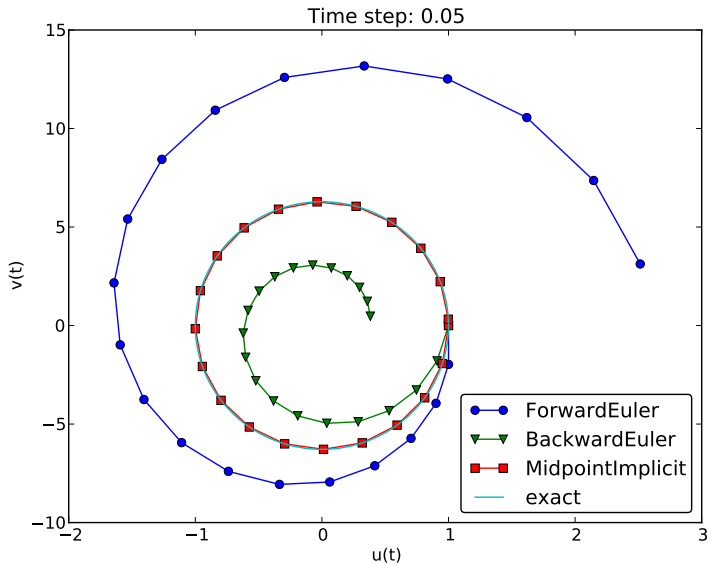
# Forward and Backward Euler and Crank-Nicolson

```
solvers = [  
    odespy.ForwardEuler(f),  
    # Implicit methods must use Newton solver to converge  
    odespy.BackwardEuler(f, nonlinear_solver='Newton'),  
    odespy.CrankNicolson(f, nonlinear_solver='Newton'),  
]
```

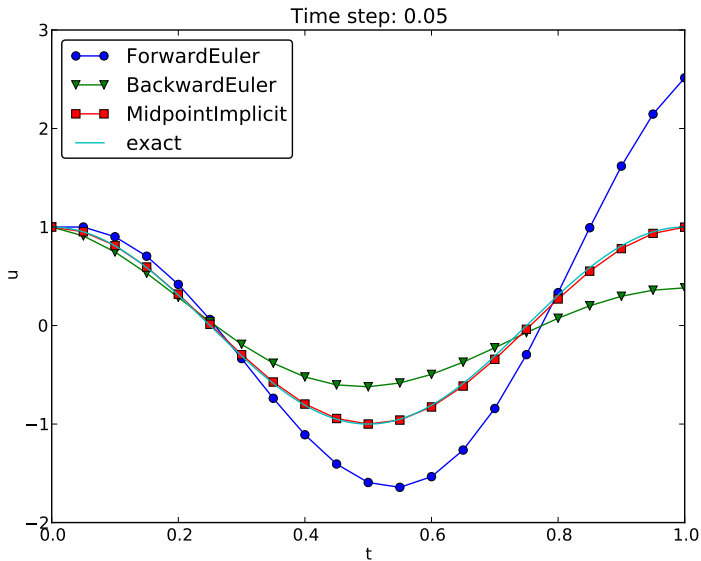
Two plot types:

- $u(t)$  vs  $t$
- Parameterized curve  $(u(t), v(t))$  in *phase space* (exact:  $(I \cos \omega t, -\omega I \sin \omega t) = \text{ellipse}$ )
- Notice: The curve  $(u(t), v(t))$  is closed and periodic ( $P = 2\pi/\omega$ )

# Phase plane plot of the numerical solutions



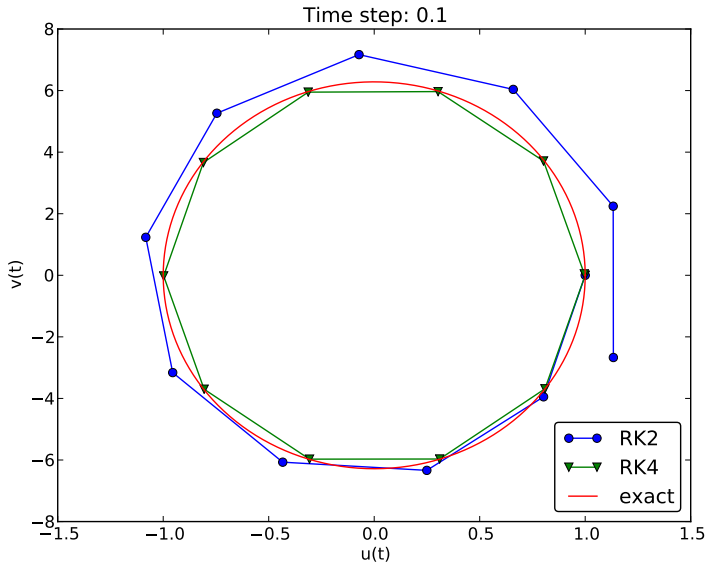
# Plain solution curves



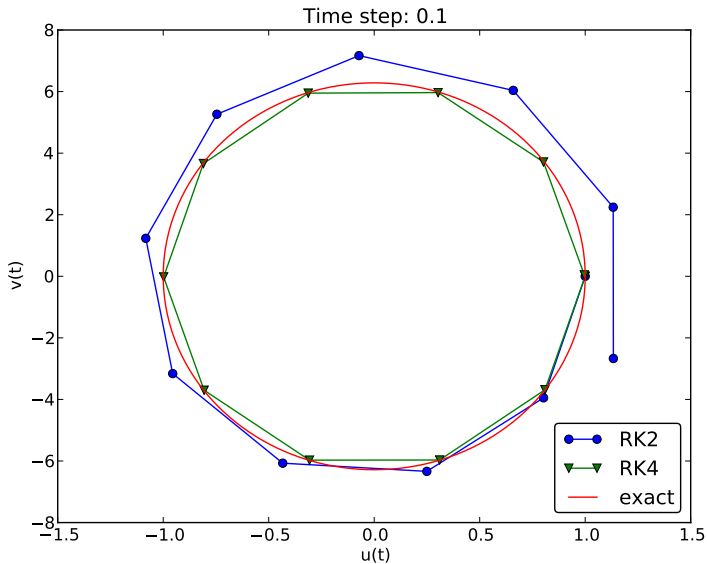
# Observations from the figures

- Forward Euler has growing amplitude and outward  $(u, v)$  spiral - pumps energy into the system.
- Backward Euler is opposite: decreasing amplitude, inward spiral, extracts energy.
- **Forward and Backward Euler are useless for vibrations.**
- Crank-Nicolson (MidpointImplicit) looks much better.

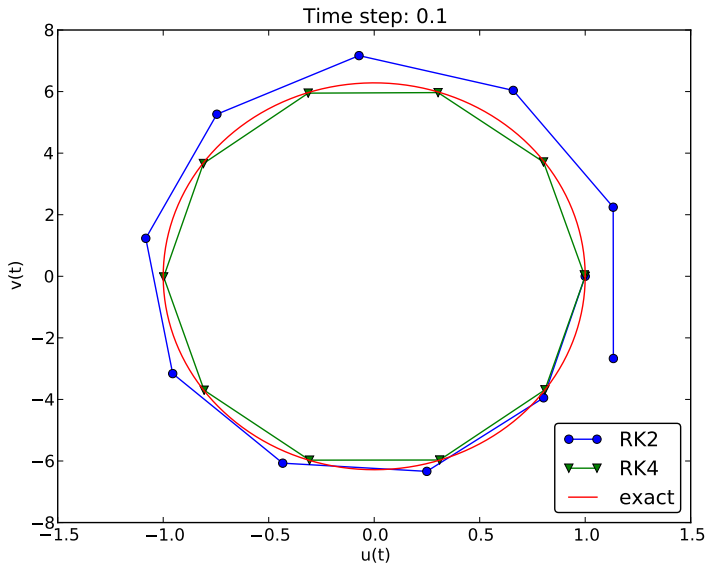
# Runge-Kutta methods of order 2 and 4; short time series



# Runge-Kutta methods of order 2 and 4; longer time series



# Crank-Nicolson; longer time series



# Observations of RK and CN methods

- 4th-order Runge-Kutta is very accurate, also for large  $\Delta t$ .
- 2th-order Runge-Kutta is almost as bad as Forward and Backward Euler.
- Crank-Nicolson is accurate, but the amplitude is not as accurate as the difference scheme for  $u'' + \omega^2 u = 0$ .



# The Euler-Cromer method; idea

Forward-backward discretization of the 2x2 system:

- Update  $u$  with Forward Euler
- Update  $v$  with Backward Euler, using latest  $u$

$$[D_t^+ u = v]^n, \quad (31)$$

$$[D_t^- v = -\omega u]^{n+1}. \quad (32)$$

# The Euler-Cromer method; complete formulas

Written out:

$$u^0 = I, \quad (33)$$

$$v^0 = 0, \quad (34)$$

$$u^{n+1} = u^n + \Delta t v^n, \quad (35)$$

$$v^{n+1} = v^n - \Delta t \omega^2 u^{n+1}. \quad (36)$$

Names: Forward-backward scheme, Semi-implicit Euler method, symplectic Euler, semi-explicit Euler, Newton-Stormer-Verlet, and Euler-Cromer.

Forward Euler and Backward Euler are both  $\mathcal{O}(\Delta t)$  approximations. What about the overall scheme? Expect  $\mathcal{O}(\Delta t)$ ...

## Equivalence with the scheme for the second-order ODE

Goal: eliminate  $v^n$ . We have

$$v^n = v^{n-1} - \Delta t \omega^2 u^n,$$

which can be inserted in (??) to yield

$$u^{n+1} = u^n + \Delta t v^{n-1} - \Delta t^2 \omega^2 u^n. \quad (37)$$

Using (??),

$$v^{n-1} = \frac{u^n - u^{n-1}}{\Delta t},$$

and when this is inserted in (??) we get

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2 \omega^2 u^n \quad (38)$$

# Comparison of the treatment of initial conditions

- The Euler-Cromer scheme is nothing but the centered scheme for  $u'' + \omega^2 u = 0$  (??)!
- The previous analysis of this scheme then also applies to the Euler-Cromer method!
- What about the initial conditions?

$$u' = v = 0 \quad \Rightarrow \quad v^0 = 0,$$

and (??) implies  $u^1 = u^0$ , while (??) says  $v^1 = -\omega^2 u^0$ .

This  $u^1 = u^0$  approximation corresponds to a first-order Forward Euler discretization of  $u'(0) = 0$ :  $[D_t^+ u = 0]^0$ .

# A method utilizing a staggered mesh

- The Euler-Cromer scheme uses two unsymmetric differences in a symmetric way...
- We can derive the method from a more pedagogical point of view where we use a *staggered mesh* and only centered differences

Staggered mesh:

- $u$  is unknown at mesh points  $t_0, t_1, \dots, t_n, \dots$
- $v$  is unknown at mesh points  $t_{1/2}, t_{3/2}, \dots, t_{n+1/2}, \dots$   
(between the  $u$  points)

## Centered differences on a staggered mesh

$$[D_t u = v]^{n+\frac{1}{2}}, \quad (39)$$

$$[D_t v = -\omega u]^{n+1}. \quad (40)$$

Written out:

$$u^{n+1} = u^n + \Delta t v^{n+\frac{1}{2}}, \quad (41)$$

$$v^{n+\frac{3}{2}} = v^{n+\frac{1}{2}} - \Delta t \omega^2 u^{n+1}. \quad (42)$$

or shift one time level back (purely of esthetic reasons):

$$u^n = u^{n-1} + \Delta t v^{n-\frac{1}{2}}, \quad (43)$$

$$v^{n+\frac{1}{2}} = v^{n-\frac{1}{2}} - \Delta t \omega^2 u^n. \quad (44)$$

# Comparison with the scheme for the 2nd-order ODE

- Can eliminate  $v^{n\pm 1/2}$  and get the centered scheme for  $u'' + \omega^2 u = 0$
- What about the initial conditions? Their equivalent too!

$u(0) = 0$  and  $u'(0) = v(0) = 0$  give  $u^0 = I$  and

$$v(0) \approx \frac{1}{2}(v^{-\frac{1}{2}} + v^{\frac{1}{2}}) = 0, \quad \Rightarrow \quad v^{-\frac{1}{2}} = -v^{\frac{1}{2}}.$$

Combined with the scheme on the staggered mesh we get

$$u^1 = u^0 - \frac{1}{2}\Delta t^2 \omega^2 I,$$

**Generalization: damping, nonlinear spring, and external excitation**

$$mu'' + f(u') + s(u) = F(t), \quad u(0) = I, \quad u'(0) = V, \quad t \in (0, T]. \quad (45)$$

Input data:  $m$ ,  $f(u')$ ,  $s(u)$ ,  $F(t)$ ,  $I$ ,  $V$ , and  $T$ .

Typical choices of  $f$  and  $s$ :

# A centered scheme for quadratic damping

- $f(u') = bu'|u'|$  leads to a quadratic equation for  $u^{n+1}$
- Instead of solving the quadratic equation, we use a geometric mean approximation

$$[mD_t D_t u + bu'|u'| + s(u) = F]^n, \quad (52)$$

Compute  $u'|u'|$  by a *geometric mean*!



# Using a geometric mean to linearize the equation

Geometric mean approximation:

$$w^n \approx w^{n-1/2} w^{n+1/2}.$$

For  $|u'|u'|$ :

$$[u'|u'|]^n \approx u'(t_n + \frac{1}{2})|u'(t_n - \frac{1}{2})|.$$

For  $u'$  at  $t_{n\pm 1/2}$  we use centered difference:

$$u'(t_{n+1/2}) \approx [D_t u]^{n+1/2}, \quad u'(t_{n-1/2}) \approx [D_t u]^{n-1/2}. \quad (53)$$

After some algebra:

$$u^{n+1} = (m + b|u^n - u^{n-1}|)^{-1} \times \\ (2mu^n - mu^{n-1} + bu^n|u^n - u^{n-1}| + \Delta t^2(F^n - s(u^n))) . \quad (54)$$

## Initial condition for quadratic damping

Simply use that  $u' = V$  in the scheme:

$$[mD_tD_t u + bV|V| + s(u) = F]^0 \quad (55)$$

which gives

$$u^1 = u^0 + \Delta t V + \frac{\Delta t^2}{2m} (-bV|V| - s(u^0) + F^0) . \quad (56)$$

The computational algorithm is a slight variation of the one in Section ??:

- ❶  $u^0 = I$
- ❷ compute  $u^1$  from (??) if linear damping or (??) if quadratic damping
- ❸ for  $n = 1, 2, \dots, N_t - 1$ :
  - ❶ compute  $u^{n+1}$  from (vib:ode2:step4) if linear damping or (??) if quadratic damping

# Implementation

```
def solver(I, V, m, b, s, F, dt, T, damping='linear'):
    """
    Solve  $m u'' + f(u') + s(u) = F(t)$  for  $t$  in  $(0, T]$ ,
     $u(0)=I$  and  $u'(0)=V$ ,
    by a central finite difference method with time step  $dt$ .
    If damping is 'linear',  $f(u')=b*u$ , while if damping is
    'quadratic',  $f(u')=b*u'*abs(u')$ .
     $F(t)$  and  $s(u)$  are Python functions.
    """
    dt = float(dt); b = float(b); m = float(m) # avoid integer div.
    Nt = int(round(T/dt))
    u = zeros(Nt+1)
    t = linspace(0, Nt*dt, Nt+1)

    u[0] = I
    if damping == 'linear':
        u[1] = u[0] + dt*V + dt**2/(2*m)*(-b*V - s(u[0]) + F(t[0]))
    elif damping == 'quadratic':
        u[1] = u[0] + dt*V + \
            dt**2/(2*m)*(-b*V*abs(V) - s(u[0]) + F(t[0]))

    for n in range(1, Nt):
        if damping == 'linear':
            u[n+1] = (2*m*u[n] + (b*dt/2 - m)*u[n-1] +
                      dt**2*(F(t[n]) - s(u[n])))/(m + b*dt/2)
        elif damping == 'quadratic':
            u[n+1] = (2*m*u[n] - m*u[n-1] + b*u[n]*abs(u[n] - u[n-1])
                      + dt**2*(F(t[n]) - s(u[n])))/\
```

- Exact linear solution of the discrete equations with linear  $s$
- MMS

# Demo program

vib.py supports input via the command line:

```
Terminal> python vib.py --s 'sin(u)' --F '3*cos(4*t)' --c 0.03
```

This results in a moving window following the function on the screen.

