

Study Guide: Vibration ODEs

Hans Petter Langtangen^{1,2}

Center for Biomedical Computing, Simula Research Laboratory¹

Department of Informatics, University of Oslo²

September 13, 2012

A simple vibration problem

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

Exact solution:

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

$u(t)$ oscillates with constant amplitude I and (angular) frequency ω . 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 $[0, T]$.
- 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:

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

Inserting (??) in (??) yields

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

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

- ❶ $u^0 = I$
- ❷ compute u^1 from (??)
- ❸ for $n = 1, 2, \dots, N_t - 1$:
 - ❶ 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: I (consistently) use w for ω 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:

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

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
```


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`.

Effect of the time step on long simulations

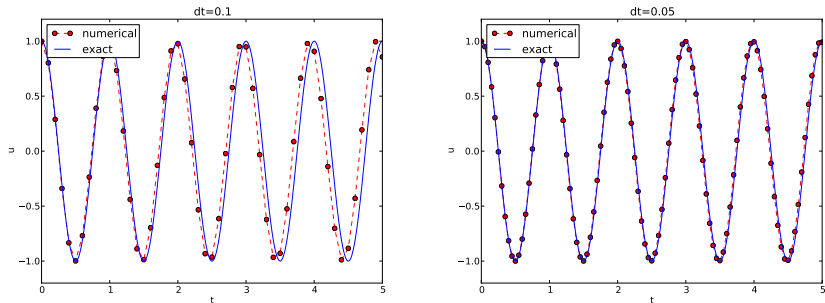


Figure: Effect of halving the time step.

Observations:

- 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: $u_e(t) = I \cos(\omega t) \sim I \exp(i\omega t) = I(\exp(i\omega\Delta t))^n$
- Trick for simplifying the algebra: $A = \exp(i\tilde{\omega}\Delta t)$ (ansatz)
- $\tilde{\omega}$: 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; calculations (1)

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

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

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 ω ?

Polynomial approximation of the phase error

Taylor series expansion for small $\omega\Delta t$ 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
```

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

Plot of the phase error

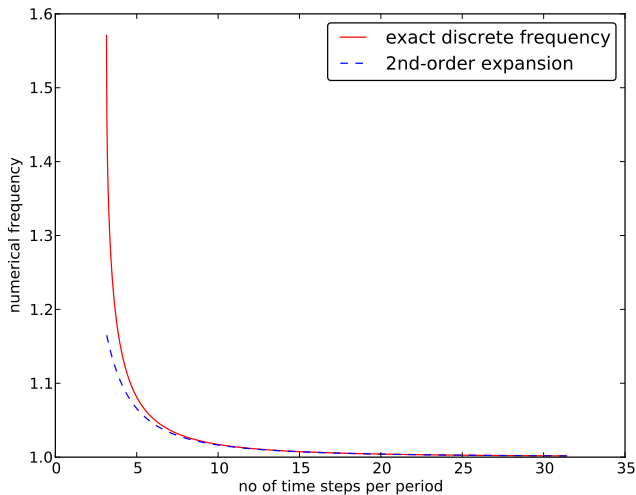


Figure: Exact discrete frequency and its second-order series expansion.

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

Ideal for verification and analysis!

Stability

- We have observed constant amplitude (desired!), but phase error.
- Constant amplitude: \sin is real-valued $\Rightarrow \tilde{\omega}$ is real-valued.
- What if $\tilde{\omega}$ is complex?
- $\sin^{-1}(x)$ is complex if $|x| > 1$.
- Complex $\tilde{\omega} = \tilde{\omega}_r + i\tilde{\omega}_i$.
- Since $\sin^{-1}(x)$ has a *negative* imaginary part for $x > 1$, it means that $\exp(i\omega\tilde{t}) = \exp(-\tilde{\omega}_i t) \exp(i\tilde{\omega}_r t)$ will lead to exponential growth in time because $\tilde{\omega}_i < 0$ and hence $-\tilde{\omega}_i t > 0$.

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

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

Figure ?? displays what happens when $\Delta t = \pi^{-1} + 9.01 \cdot 10^{-5}$ (π^{-1} is the stability limit).

Alternative schemes based on 1st-order equations

Standard technique for $u'' + \dots$ (and any higher-order ODE):
rewrite as first-order system.

Here:

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

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

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

Standard methods for 1st-order ODE systems

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* 2×2 system for the new values!

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

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 Euler, Backward Euler, and Crank-Nicolson
(MidpointImplicit in Odespy) are first out:

```
solvers = [
    odespy.ForwardEuler(f),
```

The Euler-Cromer method

Forward-backward discretization:

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

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

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

Written out:

$$u^{n+1} = u^n + \Delta t v^n, \quad (33)$$

$$v^{n+1} = v^n - \Delta t \omega^2 u^{n+1}. \quad (34)$$

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

Eliminate v^n :

From (??):

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

which can be inserted in (??) to yield

$$u^{n+1} = u^n + \Delta t v^{n-1} - \Delta t^2 \omega^2 u^n. \quad (35)$$

Using (??),

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

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

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

which is nothing but the centered scheme (??)! The previous analysis of this scheme then also applies to the Euler-Cromer method!

The initial condition $u' = 0$:

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

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