A simple vibration problem

Implementation

Verification

Long time simulations

- Long time simulations visualized with aid of Bokeh: coupled panning of multiple graphs
- How does Bokeh plotting code look like?

Analysis of the numerical scheme

Alternative schemes based on 1st-order equations

Generalization: damping, nonlinear spring, and external excitation
A simple vibration problem

\[ u''(t) + \omega^2 u = 0, \quad u(0) = l, \quad u'(0) = 0, \quad t \in (0, T] \]

Exact solution:

\[ u(t) = l \cos(\omega t) \]

\( u(t) \) oscillates with constant amplitude \( l \) and (angular) frequency \( \omega \). Period: \( P = 2\pi / \omega \).
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:
$$u''(t_n) + \omega^2 u(t_n) = 0, \quad n = 1, \ldots, N_t$$
Step 3: Approximate derivative(s) by finite difference approximation(s). Very common (standard!) formula for $u''$:

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

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

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = -\omega^2 u^n$$
Step 4: Formulate the computational algorithm. Assume $u^{n-1}$ and $u^n$ are known, solve for unknown $u^{n+1}$:

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

Nick names for this scheme: Störmer’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 $u'(0) = 0$ by a centered difference

$$\frac{u^1 - u^{-1}}{2\Delta t} = 0 \Rightarrow u^{-1} = u^1$$

Inserted in the scheme for $n = 0$ gives

$$u^1 = u^0 - \frac{1}{2} \Delta t^2 \omega^2 u^0$$
The computational algorithm

1. \( u^0 = I \)
2. compute \( u^1 \)
3. for \( n = 1, 2, \ldots, N_t - 1 \):
   1. compute \( u^{n+1} \)

More precisely expressed in Python:

```python
import numpy as np

# mesh points in time
Nt = ... # number of steps
T = ... # total time

# constant time step.
t = np.linspace(0, T, Nt+1)
dt = t[1] - t[0]

# solution
u = np.zeros(Nt+1)
u[0] = I

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

\[
[D_t D_t u + \omega^2 u = 0]^n
\]

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

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

which is written out as

\[
\frac{1}{\Delta t} \left( \frac{u^{n+1} - u^n}{\Delta t} - \frac{u^n - u^{n-1}}{\Delta t} \right) = \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2}.
\]
Operator notation; initial condition

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

where \([D_{2t} u]^n\) is defined as

\[ [D_{2t} u]^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t} . \]
$u$ is often displacement/position, $u'$ is velocity and can be computed by

$$u'(t_n) \approx \frac{u^{n+1} - u^{n-1}}{2\Delta t} = [D_{2t}u]^n$$
A simple vibration problem

Implementation

Verification

Long time simulations

- Long time simulations visualized with aid of Bokeh: coupled panning of multiple graphs
- How does Bokeh plotting code look like?

Analysis of the numerical scheme

Alternative schemes based on 1st-order equations

Generalization: damping, nonlinear spring, and external excitation
import numpy as np
import matplotlib.pyplot as plt

def solver(I, w, dt, T):
    
    Solve $u'' + w^2u = 0$ 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 = np.zeros(Nt+1)
    t = np.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

def solver_adjust_w(I, w, dt, T, adjust_w=True):
    
    Solve $u'' + w^2u = 0$ 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 = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)

    w_adj = w*(1 - w**2*dt**2/24.) if adjust_w else w
    u[0] = I
    u[1] = u[0] - 0.5*dt**2*w_adj**2*u[0]
    for n in range(1, Nt):
        u[n+1] = 2*u[n] - u[n-1] - dt**2*w_adj**2*u[n]
    return u, t
def u_exact(t, I, w):
    return I*np.cos(w*t)

def visualize(u, t, I, w):
    plt.plot(t, u, 'r--o')
    t_fine = np.linspace(0, t[-1], 1001)  # very fine mesh for u_e
    u_e = u_exact(t_fine, I, w)
    plt.plot(t_fine, u_e, 'b-')
    plt.legend(['numerical', 'exact'], loc='upper left')
    plt.xlabel('t')
    plt.ylabel('u')
    dt = t[1] - t[0]
    plt.title('dt=%g' % dt)
    umin = 1.2*u.min(); umax = -umin
    plt.axis([t[0], t[-1], umin, umax])
    plt.savefig('tmp1.png'); plt.savefig('tmp1.pdf')
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)
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
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> ffmpeg -r 12 -i tmp_vib%04d.png -c:v flv movie.flv

Can use avconv instead of ffmpeg.

<table>
<thead>
<tr>
<th>Format</th>
<th>Codec and filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash</td>
<td>-c:v flv movie.flv</td>
</tr>
<tr>
<td>MP4</td>
<td>-c:v libx264 movie.mp4</td>
</tr>
<tr>
<td>Webm</td>
<td>-c:v libvpx movie.webm</td>
</tr>
<tr>
<td>Ogg</td>
<td>-c:v libtheora movie.ogg</td>
</tr>
</tbody>
</table>
A simple vibration problem

Implementation

Verification

Long time simulations

- Long time simulations visualized with aid of Bokeh: coupled panning of multiple graphs
- How does Bokeh plotting code look like?

Analysis of the numerical scheme

Alternative schemes based on 1st-order equations

Generalization: damping, nonlinear spring, and external excitation
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.
The next function estimates convergence rates, i.e., it

- performs \( m \) simulations with halved time steps: \( 2^{-k} \Delta t \), \( k = 0, \ldots, m - 1 \),
- 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 \( E_{i-1} = C \Delta t_{i-1}^{r_i} \).
def convergence_rates(m, solver_function, 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.
    solver_function(I, w, dt, T) solves each problem, where T is based on simulation for num_periods periods.
    """
    from math import pi
    w = 0.35; I = 0.3  # just chosen values
    P = 2*pi/w          # period
    dt = P/30           # 30 time step per period 2*pi/w
    T = P*num_periods

    dt_values = []
    E_values = []
    for i in range(m):
        u, t = solver_function(I, w, dt, T)
        u_e = u_exact(t, I, w)
        E = np.sqrt(dt*np.sum((u_e-u)**2))
        dt_values.append(dt)
        E_values.append(E)
        dt = dt/2

    r = [np.log(E_values[i-1]/E_values[i])/np.log(dt_values[i-1]/dt_values[i])
         for i in range(1, m, 1)]
    return r, E_values, dt_values

Result:
r contains values equal to 2.00 - as expected!
Unit test for the convergence rate

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

```python
def test_convergence_rates():
r, E, dt = convergence_rates(
m=5, solver_function=solver, num_periods=8)
    # Accept rate to 1 decimal place
tol = 0.1
assert abs(r[-1] - 2.0) < tol
    # Test that adjusted $\omega$ obtains 4th order convergence
r, E, dt = convergence_rates(
m=5, solver_function=solver_adjust_w, num_periods=8)
print 'adjust w rates:', r
assert abs(r[-1] - 4.0) < tol

def plot_convergence_rates():
r2, E2, dt2 = convergence_rates(
m=5, solver_function=solver, num_periods=8)
plt.loglog(dt2, E2)
r4, E4, dt4 = convergence_rates(
m=5, solver_function=solver_adjust_w, num_periods=8)
plt.loglog(dt4, E4)
plt.legend(['original scheme', r'adjusted $\omega$'],
    loc='upper left')
plt.title('Convergence of finite difference methods')
from plotslopes import slope_marker
slope_marker((dt2[1], E2[1]), (2,1))
slope_marker((dt4[1], E4[1]), (4,1))
plt.savefig('tmp_convrate.png'); plt.savefig('tmp_convrate.pdf')
```

Complete code in `vib_undamped.py`.
A simple vibration problem

Implementation

Verification

Long time simulations

- Long time simulations visualized with aid of Bokeh: coupled panning of multiple graphs
- How does Bokeh plotting code look like?

Analysis of the numerical scheme

Alternative schemes based on 1st-order equations

Generalization: damping, nonlinear spring, and external excitation
The numerical solution seems to have right amplitude.

- There is an angular frequency error (reduced by reducing the time step).
- The total angular frequency 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 2: `scitools.avplotter` (ASCII vertical plotter).

Example:

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

Movie of the moving plot window.

!splot

- **Bokeh** is a Python plotting library for fancy web graphics
- Example here: long time series with many coupled graphs that can move simultaneously
1. A simple vibration problem

2. Implementation

3. Verification

4. Long time simulations
   - Long time simulations visualized with aid of Bokeh: coupled panning of multiple graphs
   - How does Bokeh plotting code look like?

5. **Analysis of the numerical scheme**

6. Alternative schemes based on 1st-order equations

7. Generalization: damping, nonlinear spring, and external excitation
Analysis of the numerical scheme

Can we understand the frequency error?

![Graphs showing numerical vs. exact solutions for different dt values](image)
Movie of the angular frequency error

\[ u'' + \omega^2 u = 0, \quad u(0) = 1, \quad u'(0) = 0, \quad \omega = 2\pi, \quad u_e(t) = \cos(2\pi t), \]
\[ \Delta t = 0.05 \text{ (20 intervals per period)} \]

Movie 1: https://raw.githubusercontent.com/hplgit/fdm-book/master/doc/.src/chapters/vib/[[mov-vib/vib_undamped_movie_dt0.05/movie.ogg]]
We can derive an exact solution of the discrete equations

- We have a linear, homogeneous, difference equation for \( u^n \).
- Has solutions \( u^n \sim lA^n \), where \( A \) is unknown (number).
- Here: \( u_e(t) = l \cos(\omega t) \sim l \exp(i\omega t) = l(e^{i\omega \Delta t})^n \)

- Trick for simplifying the algebra: \( u^n = lA^n \), with \( A = \exp(i\tilde{\omega} \Delta t) \), then find \( \tilde{\omega} \)

- \( \tilde{\omega} \): unknown *numerical frequency* (easier to calculate than \( A \))

- \( \omega - \tilde{\omega} \) is the angular *frequency error*

- Use the real part as the physical relevant part of a complex expression
Calculations of an exact solution of the discrete equations

\[ u^n = IA^n = I \exp(\tilde{\omega} \Delta t n) = I \exp(\tilde{\omega} t) = I \cos(\tilde{\omega} t) + il \sin(\tilde{\omega} t). \]

\[ [D_tD_t u]^n = \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = I \frac{A^{n+1} - 2A^n + A^{n-1}}{\Delta t^2} \]

\[ = I \frac{\exp(i\tilde{\omega}(t + \Delta t)) - 2 \exp(i\tilde{\omega}t) + \exp(i\tilde{\omega}(t - \Delta t))}{\Delta t^2} \]

\[ = I \exp(i\tilde{\omega}t) \frac{1}{\Delta t^2} (\exp(i\tilde{\omega}(\Delta t)) + \exp(i\tilde{\omega}(-\Delta t)) - 2) \]

\[ = I \exp(i\tilde{\omega}t) \frac{2}{\Delta t^2} (\cosh(i\tilde{\omega}\Delta t) - 1) \]

\[ = I \exp(i\tilde{\omega}t) \frac{2}{\Delta t^2} (\cos(\tilde{\omega}\Delta t) - 1) \]

\[ = -I \exp(i\tilde{\omega}t) \frac{4}{\Delta t^2} \sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) \]
Solving for the numerical frequency

The scheme with \( u^n = I \exp(i \omega \Delta t n) \) inserted gives

\[-I \exp(i \tilde{\omega} t) \frac{4}{\Delta t^2} \sin^2(\frac{\tilde{\omega} \Delta t}{2}) + \omega^2 I \exp(i \tilde{\omega} t) = 0\]

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

\[\frac{4}{\Delta t^2} \sin^2(\frac{\tilde{\omega} \Delta t}{2}) = \omega^2\]

Solve for \( \tilde{\omega} \):

\[\tilde{\omega} = \pm \frac{2}{\Delta t} \sin^{-1}\left(\frac{\omega \Delta t}{2}\right)\]

- Frequency error because \( \tilde{\omega} \neq \omega \).
- Note: dimensionless number \( p = \omega \Delta t \) is the key parameter (i.e., no of time intervals per period is important, not \( \Delta t \) itself)
- But how good is the approximation \( \tilde{\omega} \) to \( \omega \)?
Taylor series expansion for small $\Delta t$ gives a formula that is easier to understand:

```python
>>> 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  # note the final "/dt"
```

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

The numerical frequency is too large (too fast oscillations).
Plot of the frequency error

Recommendation: 25-30 points per period.
Exact discrete solution

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

The error mesh function,

$$e^n = u_e(t_n) - u^n = l \cos (\omega n\Delta t) - l \cos (\tilde{\omega} n\Delta t)$$

is ideal for verification and further analysis!

$$e^n = l \cos (\omega n\Delta t) - l \cos (\tilde{\omega} n\Delta t) = -2l \sin \left( t \frac{1}{2} (\omega - \tilde{\omega}) \right) \sin \left( t \frac{1}{2} (\omega + \tilde{\omega}) \right)$$
Can easily show convergence:

\[ e^n \to 0 \text{ as } \Delta t \to 0, \]

because

\[ \lim_{\Delta t \to 0} \tilde{\omega} = \lim_{\Delta t \to 0} \frac{2}{\Delta t} \sin^{-1} \left( \frac{\omega \Delta t}{2} \right) = \omega, \]

by L’Hopital’s rule or simply asking sympy: or WolframAlpha:

```python
>>> import sympy as sym
>>> dt, w = sym.symbols('x w')
>>> sym.limit((2/dt)*sym.asin(w*dt/2), dt, 0, dir='+')
w
```
Observations:

- Numerical solution has constant amplitude (desired!), but an angular frequency 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 $-\tilde{\omega}_it > 0$
- This is *instability* because the qualitative behavior is wrong
The stability criterion

Cannot tolerate growth and must therefore demand a stability criterion

$$\frac{\omega \Delta t}{2} \leq 1 \quad \Rightarrow \quad \Delta t \leq \frac{2}{\omega}$$

Try $\Delta t = \frac{2}{\omega} + 9.01 \cdot 10^{-5}$ (slightly too big!):
Summary of the analysis

We can draw three important conclusions:

1. The key parameter in the formulas is \( p = \omega \Delta t \) (dimensionless)
   - Period of oscillations: \( P = \frac{2\pi}{\omega} \)
   - Number of time steps per period: \( N_P = \frac{P}{\Delta t} \)
   - \( \Rightarrow \ p = \omega \Delta t = \frac{2\pi}{N_P} \sim \frac{1}{N_P} \)
   - The smallest possible \( N_P \) is 2 \( \Rightarrow \ p \in (0, \pi] \)

2. For \( p \leq 2 \) the amplitude of \( u^n \) is constant (stable solution)

3. \( u^n \) has a relative frequency error \( \tilde{\omega}/\omega \approx 1 + \frac{1}{24} p^2 \), making numerical peaks occur too early
1. A simple vibration problem

2. Implementation

3. Verification

4. Long time simulations
   - Long time simulations visualized with aid of Bokeh: coupled panning of multiple graphs
   - How does Bokeh plotting code look like?

5. Analysis of the numerical scheme

6. Alternative schemes based on 1st-order equations

7. Generalization: damping, nonlinear spring, and external excitation
The vast collection of ODE solvers (e.g., in Odespy) cannot be applied to
\[ u'' + \omega^2 u = 0 \]
unless we write this higher-order ODE as a system of 1st-order ODEs.

Introduce an auxiliary variable \( v = u' \):

\[ u' = v, \]
\[ v' = -\omega^2 u. \]

Initial conditions: \( u(0) = 1 \) and \( v(0) = 0 \).
We apply the Forward Euler scheme to each component equation:

\[ D_t^+ u = v ]^n, \]
\[ D_t^+ v = -\omega^2 u ]^n, \]

or written out,

\[ u^{n+1} = u^n + \Delta t v^n, \]
\[ v^{n+1} = v^n - \Delta t \omega^2 u^n. \]
We apply the Backward Euler scheme to each component equation:

\[ D_t^- u = -\omega u \] \[ \text{and} \] \[ D_t^- v = v \] \[ n+1, \] \[ n+1. \] \[ (5) \] \[ (6) \]

Written out:

\[ u^{n+1} - \Delta t v^{n+1} = u^n, \] \[ (7) \]
\[ v^{n+1} + \Delta t \omega^2 u^{n+1} = v^n. \] \[ (8) \]

This is a coupled 2 × 2 system for the new values at \( t = t_{n+1} \)!
The Crank-Nicolson scheme

\[ [D_t u = \bar{v}^t]^{n + \frac{1}{2}}, \]  
\[ (9) \]
\[ [D_t v = -\omega \bar{u}^t]^{n + \frac{1}{2}}. \]  
\[ (10) \]

The result is also a coupled system:

\[ u^{n+1} - \frac{1}{2} \Delta t v^{n+1} = u^n + \frac{1}{2} \Delta t v^n, \]  
\[ (11) \]
\[ v^{n+1} + \frac{1}{2} \Delta t \omega^2 u^{n+1} = v^n - \frac{1}{2} \Delta t \omega^2 u^n. \]  
\[ (12) \]
Can use **Odespy** to compare many methods for first-order schemes:

```python
import odespy
import numpy as np

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

def run_solvers_and_plot(solvers, timesteps_per_period=20,
                          num_periods=1, I=1, w=2*np.pi):
    P = 2*np.pi/w  # duration of 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([0, I])
u, t = solver.solve(t_mesh)
```
Forward and Backward Euler and Crank-Nicolson

```python
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 curve is an ellipse: $(l \cos \omega t, -\omega l \sin \omega t)$, closed and periodic
Phase plane plot of the numerical solutions

Note: CrankNicolson in Odespy leads to the name MidpointImplicit in plots.
Figure: Comparison of classical schemes.
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 (Midpoint Implicit) 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

Time step: 0.1
- \( u(t) \)
- \( v(t) \)
- RK2
- RK4
- exact

Time step: 0.05
- \( u(t) \)
- \( v(t) \)
- RK2
- RK4
- exact
Crank-Nicolson; longer time series

(MidpointImplicit means CrankNicolson in Odespy)
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 model

$$u'' + \omega^2 u = 0, \quad u(0) = I, \quad u'(0) = V,$$

has the nice energy conservation property that

$$E(t) = \frac{1}{2} (u')^2 + \frac{1}{2} \omega^2 u^2 = \text{const}.$$

This can be used to check solutions.
Derivation of the energy conservation property

Multiply $u'' + \omega^2 u = 0$ by $u'$ and integrate:

$$\int_0^T u'' u' dt + \int_0^T \omega^2 uu' dt = 0.$$

Observing that

$$u'' u' = \frac{d}{dt} \frac{1}{2} (u')^2, \quad uu' = \frac{d}{dt} \frac{1}{2} u^2,$$

we get

$$\int_0^T \left( \frac{d}{dt} \frac{1}{2} (u')^2 + \frac{d}{dt} \frac{1}{2} \omega^2 u^2 \right) dt = E(T) - E(0),$$

where

$$E(t) = \frac{1}{2} (u')^2 + \frac{1}{2} \omega^2 u^2$$
Remark about $E(t)$

$E(t)$ does not measure energy, energy per mass unit.

Starting with an ODE coming directly from Newton’s 2nd law $F = ma$ with a spring force $F = -ku$ and $ma = mu''$ (a: acceleration, $u$: displacement), we have

$$mu'' + ku = 0$$

Integrating this equation gives a physical energy balance:

$$E(t) = \frac{1}{2}mv^2 + \frac{1}{2}ku^2 = E(0), \quad v = u'$$

kinetic energy  potential energy

Note: the balance is not valid if we add other terms to the ODE.
The Euler-Cromer method; idea

2x2 system for $u'' + \omega^2 u = 0$:

$v' = -\omega^2 u$
$u' = v$

Forward-backward discretization:

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

$$[D_t^+ v = -\omega^2 u]^n$$  \hspace{1cm} (13)
$$[D_t^- u = v]^{n+1}$$  \hspace{1cm} (14)
The Euler-Cromer method; complete formulas

Written out:

\[ u^0 = l, \quad (15) \]
\[ v^0 = 0, \quad (16) \]
\[ v^{n+1} = v^n - \Delta t \omega^2 u^n \quad (17) \]
\[ u^{n+1} = u^n + \Delta t v^{n+1} \quad (18) \]

Euler-Cromer is equivalent to the scheme for $u'' + \omega^2 u = 0$

- Forward Euler and Backward Euler have error $O(\Delta t)$
- What about the overall scheme? Expect $O(\Delta t)\ldots$

We can eliminate $v^n$ and $v^{n+1}$, resulting in

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

which is the centered finite difference scheme for $u'' + \omega^2 u = 0!$
The schemes are not equivalent wrt the initial conditions

\[ u' = v = 0 \implies v^0 = 0, \]

so

\[ v^1 = v^0 - \Delta t \omega^2 u^0 = -\Delta t \omega^2 u^0 \]

\[ u^1 = u^0 + \Delta t v^1 = u^0 - \Delta t \omega^2 u^0! = \quad \underbrace{u^0 - \frac{1}{2} \Delta t \omega^2 u^0} \]

from \([D_t D_t u + \omega^2 u = 0]^n\) and \([D_{2t} u = 0]^0\)

The exact discrete solution derived earlier does not fit the Euler-Cromer scheme because of mismatch for \(u^1\).
1 A simple vibration problem
2 Implementation
3 Verification
4 Long time simulations
   - Long time simulations visualized with aid of Bokeh: coupled panning of multiple graphs
   - How does Bokeh plotting code look like?
5 Analysis of the numerical scheme
6 Alternative schemes based on 1st-order equations
7 Generalization: damping, nonlinear spring, and external excitation
Generalization: damping, nonlinear spring, and external excitation

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

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

Typical choices of \( f \) and \( s \):

- linear damping \( f(u') = bu \), or
- quadratic damping \( f(u') = bu'|u'| \)
- linear spring \( s(u) = cu \)
- nonlinear spring \( s(u) \sim \sin(u) \) (pendulum)
A centered scheme for linear damping

\[ [mD_t D_t u + f(D_{2t} u) + s(u) = F]^n \]

Written out

\[ m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + f\left(\frac{u^{n+1} - u^{n-1}}{2\Delta t}\right) + s(u^n) = F^n \]

Assume \( f(u') \) is linear in \( u' = v \):

\[ u^{n+1} = \left(2mu^n + \left(\frac{b}{2}\Delta t - m\right)u^{n-1} + \Delta t^2(F^n - s(u^n))\right)(m + \frac{b}{2}\Delta t)^{-1} \]
Initial conditions

\[ u(0) = l, \; u'(0) = V:\]

\[
\begin{align*}
[u = l]^0 & \Rightarrow u^0 = l \\
\left[D_2 t u = V\right]^0 & \Rightarrow u^{-1} = u^1 - 2\Delta t V
\end{align*}
\]

End result:

\[
u^1 = u^0 + \Delta t V + \frac{\Delta t^2}{2m}(-bV - s(u^0) + F^0)
\]

Same formula for \( u^1 \) as when using a centered scheme for \( u'' + \omega^2 u = 0. \)
\( f(u') = bu'|u'| \) leads to a quadratic equation for \( u^{n+1} \)

- Instead of solving the quadratic equation, we use a geometric mean approximation

In general, the geometric mean approximation reads

\[
(w^2)^n \approx w^{n-\frac{1}{2}} w^{n+\frac{1}{2}}.
\]

For \( |u'|u'| \) at \( t_n \):

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

For \( u' \) at \( t_{n\pm\frac{1}{2}} \) we use centered difference:

\[
u'(t_{n+\frac{1}{2}}) \approx [D_t u]^{n+\frac{1}{2}}, \quad u'(t_{n-\frac{1}{2}}) \approx [D_t u]^{n-\frac{1}{2}}
\]
A centered scheme for quadratic damping

After some algebra:

\[ u^{n+1} = (m + b|u^n - u^{n-1}|)^{-1} \times \]
\[ (2mu^n - mu^{n-1} + bu^n|u^n - u^{n-1}| + \Delta t^2(F^n - s(u^n))) \]
Simply use that \( u' = V \) in the scheme when \( t = 0 \) \((n = 0)\):

\[
[mD_t D_t u + bV|V| + s(u) = F]^0
\]

which gives

\[
u^1 = u^0 + \Delta t V + \frac{\Delta t^2}{2m} \left( -bV|V| - s(u^0) + F^0 \right)
\]
Algorithm

1. \( u^0 = l \)
2. compute \( u^1 \) (formula depends on linear/quadratic damping)
3. for \( n = 1, 2, \ldots, N_t - 1 \):
   1. compute \( u^{n+1} \) from formula (depends on linear/quadratic damping)
def solver(I, V, m, b, s, F, dt, T, damping='linear'):
    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]))) /(
                        m + b*abs(u[n] - u[n-1]))

    return u, t
Constant solution $u_e = I (V = 0)$ fulfills the ODE problem and the discrete equations. Ideal for debugging!

Linear solution $u_e = Vt + I$ fulfills the ODE problem and the discrete equations.

Quadratic solution $u_e = bt^2 + Vt + I$ fulfills the ODE problem and the discrete equations with linear damping, but not for quadratic damping. A special discrete source term can allow $u_e$ to also fulfill the discrete equations with quadratic damping.
**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.

![Graph showing a moving window following a function](image-url)
Euler-Cromer formulation

We rewrite

\[ mu'' + f(u') + s(u) = F(t), \quad u(0) = l, \quad u'(0) = V, \quad t \in (0, T] \]

as a first-order ODE system

\[ u' = v \]
\[ v' = m^{-1} (F(t) - f(v) - s(u)) \]
Staggered grid

- $u$ is unknown at $t_n$: $u^n$
- $v$ is unknown at $t_{n+1/2}$: $v^{n+\frac{1}{2}}$
- All derivatives are approximated by centered differences

$$[D_t u = v]^{n-\frac{1}{2}}$$
$$[D_t v = m^{-1} (F(t) - f(v) - s(u))]^n$$

Written out,

$$\frac{u^n - u^{n-1}}{\Delta t} = v^{n-\frac{1}{2}}$$
$$\frac{v^{n+\frac{1}{2}} - v^{n-\frac{1}{2}}}{\Delta t} = m^{-1} (F^n - f(v^n) - s(u^n))$$

Problem: $f(v^n)$
Linear damping

With \( f(v) = bv \), we can use an arithmetic mean for \( bv^n \) a la Crank-Nicolson schemes.

\[
\begin{align*}
  u^n &= u^{n-1} + \Delta t v^{n-\frac{1}{2}}, \\
  v^{n+\frac{1}{2}} &= \left(1 + \frac{b}{2m\Delta t}\right)^{-1} \left(v^{n-\frac{1}{2}} + \Delta tm^{-1} \left(F^n - \frac{1}{2}f(v^{n-\frac{1}{2}}) - s(u^n)\right)\right)
\end{align*}
\]
Quadratic damping

With \( f(\nu) = b|\nu|\nu \), we can use a geometric mean

\[
|\nu^n|\nu^n \approx |\nu^{n-\frac{1}{2}}|\nu^{n+\frac{1}{2}},
\]

resulting in

\[
\begin{align*}
u^n &= \nu^{n-1} + \Delta t\nu^{n-\frac{1}{2}}, \\
\nu^{n+\frac{1}{2}} &= (1 + \frac{b}{m}|\nu^{n-\frac{1}{2}}|\Delta t)^{-1} \left(\nu^{n-\frac{1}{2}} + \Delta t m^{-1} (\nu^n - s(\nu^n))\right).
\end{align*}
\]
Initial conditions

\[ u^0 = I \]

\[ v^\frac{1}{2} = V - \frac{1}{2} \Delta t \omega^2 I \]