Finite difference methods for vibration problems

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Oct 5, 2013

Note: **PRELIMINARY VERSION** (expect typos)

Contents

ıist of Exercises and Problems

Vibration problems lead to differential equations with solutions that o in time, typically in a damped or undamped sinusoidal fashion. Such solut certain demands on the numerical methods compared to other phenomen solutions are monotone. Both the frequency and amplitude of the osci need to be accurately handled by the numerical schemes. Most of the re and specific building blocks introduced in the fortcoming text can be to construct sound methods for partial differential equations of wave n multiple spatial dimensions.

1 Finite difference discretization

Much of the numerical challenges with computing oscillatory solutions i and PDEs can be captured by the very simple ODE $u'' + u = 0$ a is therefore the starting point for method development, implementati analysis.

1.1 A basic model for vibrations

A system that vibrates without damping and external forcing can be d by ODE problem

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

Here, ω and I are given constants. The exact solution of (1) is

$$u(t) = I\cos(\omega t)\,.$$

That is, u oscillates with constant amplitude I and angular frequency corresponding period of oscillations (i.e., the time between two neig peaks in the cosine function) is $P = 2\pi/\omega$. The number of periods per is $f = \omega/(2\pi)$ and measured in the unit Hz. Both f and ω are referr frequency, but ω may be more precisely named angular frequency, meas rad/s.

In vibrating mechanical systems modeled by (1), $u(t)$ very often repr position or a displacement of a particular point in the system. The derivat then has the interpretation of the point's velocity, and $u''(t)$ is the ass acceleration. The model (1) is not only applicable to vibrating mec systems, but also to oscillations in electrical circuits.

1.2 A centered finite difference scheme

To formulate a finite difference method for the model problem (1) we fo four steps from Section **??** in [**?**].

Step 1: Discretizing the domain. The domain is discretized by intr a uniformly partitioned time mesh in the present problem. The point mesh are hence $t_n = n\Delta t$, $n = 0, 1, \ldots, N_t$, where $\Delta t = T/N_t$ is the c length of the time steps. We introduce a mesh function u^n for $n = 0, 1$ which approximates the exact solution at the mesh points. The mesh f will be computed from algebraic equations derived from the differential e problem.

tep 2: Fulfilling the equation at discrete time points. The ODE is to e satisfied at each mesh point:

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

tep 3: Replacing derivatives by finite differences. The derivative $''(t_n)$ is to be replaced by a finite difference approximation. A common second-rder accurate approximation to the second-order derivative is

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

ıserting (4) in (3) yields

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = -\omega^2 u^n\,. \tag{5}$$

We also need to replace the derivative in the initial condition by a finite ifference. Here we choose a centered difference, whose accuracy is similar to ıe centered difference we used for u'':

$$\frac{u^1 - u^{-1}}{2\Delta t} = 0\,. \tag{6}$$

tep 4: Formulating a recursive algorithm. To formulate the computa-onal algorithm, we assume that we have already computed u^{n-1} and u^n such ıat u^{n+1} is the unknown value, which we can readily solve for:

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2\omega^2 u^n\,. \tag{7}$$

he computational algorithm is simply to apply (7) successively for $n = 2, \ldots, N_t - 1$. This numerical scheme sometimes goes under the name Störmer's ıethod or Verlet integration[1].

:omputing the first step. We observe that (7) cannot be used for $n = 0$ nce the computation of u^1 then involves the undefined value u^{-1} at $t = -\Delta t$. he discretization of the initial condition then come to rescue: (6) implies $^{-1} = u^1$ and this relation can be combined with (7) for $n = 1$ to yield a value ɔr u^1:

$$u^1 = 2u^0 - u^1 - \Delta t^2\omega^2 u^0,$$

hich reduces to

$$u^1 = u^0 - \frac{1}{2}\Delta t^2\omega^2 u^0\,. \tag{8}$$

xercise 5 asks you to perform an alternative derivation and also to generalize ıe initial condition to $u'(0) = V \neq 0$.

[1] http://en.wikipedia.org/wiki/Velocity_Verlet

The computational algorithm. The steps for solving (1) becomes

1. $u^0 = I$
2. compute u^1 from (8)
3. for $n = 1, 2, \ldots, N_t - 1$:
   (a) compute u^{n+1} from (7)

The algorithm is more precisely expressed directly 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]
```

Remark.
In the code, we use `w` as the symbol for ω. The reason is that this a prefers `w` for readability and comparison with the mathematical ω in of the full word `omega` as variable name.

Operator notation. We may write the scheme using the compact di notation (see Section **??** in [**?**]). The difference (4) has the operator ı $[D_t D_t u]^n$ such that we can write:

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

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

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

which is written out as

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

The discretization of initial conditions can in the operator nota expressed as

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

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

$$[D_{2t} u]^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t}\,.$$

Implementation

.1 Making a solver function

he algorithm from the previous section is readily translated to a complete ython function for computing (returning) $u^0, u^1, \ldots, u^{N_t}$ and $t_0, t_1, \ldots, t_{N_t}$, iven the input I, ω, Δt, and T:

```
rom numpy import *
rom matplotlib.pyplot import *
rom vib_empirical_analysis import minmax, periods, amplitudes

lef 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
```

A function for plotting the numerical and the exact solution is also convenient o have:

```
lef exact_solution(t, I, w):
    return I*cos(w*t)

lef 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')
```

corresponding main program calling these functions for a simulation of a given umber of periods (`num_periods`) may take the form


```
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)
```

Adjusting some of the input parameters on the command line can b Here is a code segment using the `ArgumentParser` tool in the `argparse` to define option value (`--option value`) pairs on the 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
```

A typical execution goes like

```
Terminal> python vib_undamped.py --num_periods 20 --dt 0.1
```

Computing u'. In mechanical vibration applications one is often int in computing the velocity $v(t) = u'(t)$ after $u(t)$ has been computed. T be done by a central difference,

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

This formula applies for all inner mesh points, $n = 1, \ldots, N_t - 1$. For n have that $v(0)$ is given by the initial condition on $u'(0)$, and for $n = N_t$ use a one-sided, backward difference: $v^n = [D_t^- u]^n$.

Appropriate vectorized Python code becomes

```
v = np.zeros_like(u)
v[1:-1] = (u[2:] - u[:-2])/(2*dt)  # internal mesh points
v[0] = V                           # Given boundary condition u'(
v[-1] = (u[-1] - u[-2])/dt         # backward difference
```

2.2 Verification

Manual calculation. The simplest type of verification, which is also tive for understanding the algorithm, is to compute u^1, u^2, and u^3 with of a calculator and make a function for comparing these results with thc the `solver` function. We refer to the `test_three_steps` function in `vib_undamped.py`[2] for details.

[2] `http://tinyurl.com/jvzzcfn/vib/vib_undamped.py`

esting very simple solutions. Constructing test problems where the exact olution is constant or linear helps initial debugging and verification as one xpects any reasonable numerical method to reproduce such solutions to machine recision. Second-order accurate methods will often also reproduce a quadratic olution. Here $[D_tD_tt^2]^n = 2$, which is the exact result. A solution $u = t^2$ leads o $u'' + \omega^2 u = 2 + (\omega t)^2 \neq 0$. We must therefore add a source in the equation: $'' + \omega^2 u = f$ to allow a solution $u = t^2$ for $f = (\omega t)^2$. By simple insertion e can show that the mesh function $u^n = t_n^2$ is also a solution of the discrete quations. Problem 1 asks you to carry out all details with showing that linear nd quadratic solutions are solutions of the discrete equations. Such results are ery useful for debugging and verification.

hecking convergence rates. Empirical computation of convergence rates, s explained in Section ?? in [?], yields a good method for verification. The unction below

- performs m simulations with halved time steps: $2^{-i}\Delta t$, $i = 0, \ldots, m-1$,
- computes the L^2 norm of the error, $E = \sqrt{2^{-i}\Delta t \sum_{n=0}^{N_t-1}(u^n - u_{\mathrm{e}}(t_n))^2}$ in each case,
- estimates the convergence rates r_i based on 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 $E_{i-1} = C\Delta t_{i-1}^{r_i}$. From these equations it follows that $r_{i-1} = \ln(E_{i-1}/E_i)/\ln(\Delta t_{i-1}/\Delta t_i)$, for $i = 1, \ldots, m-1$.

ll the implementational details appear below.

```
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
```

he returned `r` list has its values equal to 2.00, which is in excellent agreement ith what is expected from the second-order finite difference approximation

$[D_tD_tu]^n$ and other theoretical measures of the error in the numerical The final `r[-1]` value is a good candidate for 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)
```

The complete code appears in the file `vib_undamped.py`.

3 Long time simulations

Figure 1 shows a comparison of the exact and numerical solution for $\Delta t =$ and $w = 2\pi$. From the plot we make the following observations:

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

By phase error we mean that the peaks of the numerical solution have i positions compared with the peaks of the exact cosine solution. Th can be understood as if also the numerical solution is on the form I cos where $\tilde{\omega}$ is not exactly equal to ω. Later, we shall mathematically quan numerical frequency $\tilde{\omega}$.

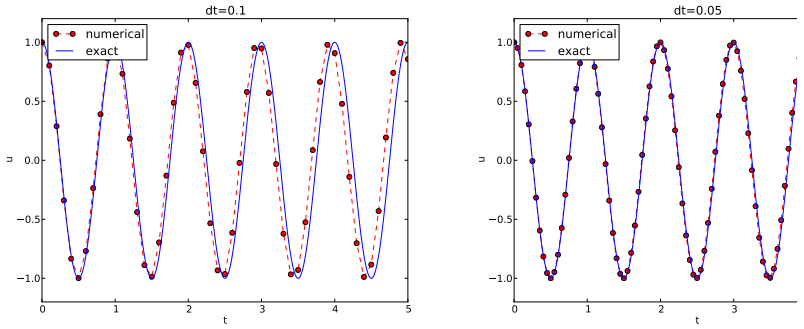


Figure 1: Effect of halving the time step.

3.1 Using a moving plot window

In vibration problems it is often of interest to investigate the system's b over long time intervals. Errors in the phase may then show up as cruc us investigate long time series by introducing a moving plot window t move along with the p most recently computed periods of the solution. Tools[3] package contains a convenient tool for this: `MovingPlotWindow`.

[3] `http://code.google.com/p/scitools`

ydoc scitools.MovingPlotWindow shows a demo and description of usage. he function below illustrates the usage and is invoked in the vib_undamped.py ode if the number of periods in the simulation exceeds 10:

```
def visualize_front(u, t, I, w, savefig=False):
    """
    Visualize u and the exact solution vs t, using a
    moving plot window and continuous drawing of the
    curves as they evolve in time.
    Makes it easy to plot very long time series.
    """
    import scitools.std as st
    from scitools.MovingPlotWindow import MovingPlotWindow

    P = 2*pi/w  # one period
    umin = 1.2*u.min();  umax = -umin
    plot_manager = MovingPlotWindow(
        window_width=8*P,
        dt=t[1]-t[0],
        yaxis=[umin, umax],
        mode='continuous drawing')
    for n in range(1,len(u)):
        if plot_manager.plot(n):
            s = plot_manager.first_index_in_plot
            st.plot(t[s:n+1], u[s:n+1], 'r-1',
                    t[s:n+1], I*cos(w*t)[s:n+1], 'b-1',
                    title='t=%6.3f' % t[n],
                    axis=plot_manager.axis(),
                    show=not savefig) # drop window if savefig
            if savefig:
                filename = 'tmp_vib%04d.png' % n
                st.savefig(filename)
                print 'making plot file', filename, 'at t=%g' % t[n]
        plot_manager.update(n)
```

Running

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

akes the simulation last for 40 periods of the cosine function. With the moving lot window we can follow the numerical and exact solution as time progresses, nd we see from this demo that the phase error is small in the beginning, but then ecomes more prominent with time. Running `vib_undamped.py` with $\Delta t = 0.1$ early shows that the phase errors become significant even earlier in the time eries and destroys the solution.

.2 Making a movie file

he `visualize_front` function stores all the plots in files whose names are umbered: `tmp_vib0000.png`, `tmp_vib0001.png`, `tmp_vib0002.png`, and so on. rom these files we may make a movie. The Flash format is popular,


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

The `avconv` program can be replaced by the `ffmpeg` program in th command if desired. Other formats can be generated by changing th codec and equipping the movie file with the right extension:

Format	Codec and filename
Flash	`-vcodec flv movie.flv`
MP4	`-vcodec libx64 movie.mp4`
Webm	`-vcodec libvpx movie.webm`
Ogg	`-vcodec libtheora movie.ogg`

The movie file can be played by some video player like `vlc`, `mplayer`, g `totem`, e.g.,

```
Terminal> vlc movie.webm
```

A web page can also be used to play the movie. Today's standard is to HTML5 `video` tag:

```
<video autoplay loop controls
       width='640' height='365' preload='none'>
<source src='movie.webm'  type='video/webm; codecs="vp8, vorbis"'
</video>
```

Caution: number the plot files correctly.
To ensure that the individual plot frames are shown in correct ord is important to number the files with zero-padded numbers (0000, 0002, etc.). The printf format `%04d` specifies an integer in a field of wi padded with zeros from the left. A simple Unix wildcard file specific like `tmp_vib*.png` will then list the frames in the right order. If the nu in the filenames were not zero-padded, the frame `tmp_vib11.png` appear before `tmp_vib2.png` in the movie.

3.3 Using a line-by-line ascii plotter

Plotting functions vertically, line by line, in the terminal window using as acters only is a simple, fast, and convenient visualization technique for lo series (the time arrow points downward). The tool `scitools.avplotter.` makes it easy to create such plots:

```
def visualize_front_ascii(u, t, I, w, fps=10):
    """
    Plot u and the exact solution vs t line by line in a
    terminal window (only using ascii characters).
```

```
    Makes it easy to plot very long time series.
    """
    from scitools.avplotter import Plotter
    import time
    P = 2*pi/w
    umin = 1.2*u.min();  umax = -umin

    p = Plotter(ymin=umin, ymax=umax, width=60, symbols='+o')
    for n in range(len(u)):
        print p.plot(t[n], u[n], I*cos(w*t[n])), \
              '%.1f' % (t[n]/P)
        time.sleep(1/float(fps))

if __name__ == '__main__':
    main()
```

he call `p.plot` returns a line of text, with the t axis marked and a symbol `+` for ıe first function (`u`) and `o` for the second function (the exact solution). Here we ppend this text a time counter reflecting how many periods the current time oint corresponds to. A typical output ($\omega = 2\pi$, $\Delta t = 0.05$) looks like this:

```
                                |                          o+      14.0
                                |                         + o      14.0
                                |                    +     o      14.1
                                |                +     o         14.1
                                |     +           o              14.2
                               +|       o                       14.2
                          +     |                                14.2
                    +       o   |                                14.3
               +     o          |                                14.4
             +   o              |                                14.4
            +o                  |                                14.5
            o +                 |                                14.5
             o    +             |                                14.6
                o      +        |                                14.6
                     o       +  |                                14.7
                          o     | +                              14.7
                                |         +                      14.8
                                |        o       +               14.8
                                |                o     +         14.9
                                |                      o   +     14.9
                                |                          o+    15.0
```

.4 Empirical analysis of the solution

or oscillating functions like those in Figure 1 we may compute the amplitude nd frequency (or period) empirically. That is, we run through the discrete olution points (t_n, u_n) and find all maxima and minima points. The distance etween two consecutive maxima (or minima) points can be used as estimate of ıe local period, while half the difference between the u value at a maximum nd a nearby minimum gives an estimate of the local amplitude.

The local maxima are the points where

$$u^{n-1} < u^n > u^{n+1}, \quad n = 1, \ldots, N_t - 1, \tag{13}$$

nd the local minima are recognized by

$$u^{n-1} > u^n < u^{n+1}, \quad n = 1, \ldots, N_t - 1\,. \tag{14}$$

ı computer code this becomes


```
def minmax(t, u):
    minima = []; maxima = []
    for n in range(1, len(u)-1, 1):
        if u[n-1] > u[n] < u[n+1]:
            minima.append((t[n], u[n]))
        if u[n-1] < u[n] > u[n+1]:
            maxima.append((t[n], u[n]))
    return minima, maxima
```

Note that the returned objects are list of tuples.

Let (t_i, e_i), $i = 0, \ldots, M-1$, be the sequence of all the M maxima where t_i is the time value and e_i the corresponding u value. The local can be defined as $p_i = t_{i+1} - t_i$. With Python syntax this reads

```
def periods(maxima):
    p = [extrema[n][0] - maxima[n-1][0]
         for n in range(1, len(maxima))]
    return np.array(p)
```

The list `p` created by a list comprehension is converted to an array s probably want to compute with it, e.g., find the corresponding freq `2*pi/p`.

Having the minima and the maxima, the local amplitude can be ca as the difference between two neighboring minimum and maximum poi

```
def amplitudes(minima, maxima):
    a = [(abs(maxima[n][1] - minima[n][1]))/2.0
         for n in range(min(len(minima),len(maxima)))]
    return np.array(a)
```

The code segments are found in the file `vib_empirical_analysis.py`[4]

Visualization of the periods `p` or the amplitudes `a` it is most conv done with just a counter on the horizontal axis, since `a[i]` and `p[i]` cor to the i-th amplitude estimate and the i-th period estimate, respectively is no unique time point associated with either of these estimate since v two different time points were used in the computations.

In the analysis of very long time series, it is advantageous to compute ı `p` and `a` instead of u to get an impression of the development of the osc

4 Analysis of the numerical scheme

4.1 Deriving an exact numerical solution

After having seen the phase error grow with time in the previous sect shall now quantify this error through mathematical analysis. The key the analysis will be to establish an exact solution of the discrete eq The difference equation (7) has constant coefficients and is homogeneo solution is then of the form $u^n = A^n$, where A is some number to be det (recall that n in u^n is a superscript labeling the time level, while n

[4] `http://tinyurl.com/jvzzcfn/vib/vib_empirical_analysis.py`

an exponent). With oscillating functions as solutions, the algebra will be onsiderably simplified if we write

$$A = Ie^{i\tilde{\omega}\Delta t},$$

nd solve for the numerical frequency $\tilde{\omega}$ rather than A. Note that $i = \sqrt{-1}$ is the naginary unit. (Using a complex exponential function gives simpler arithmetics ıan working with a sine or cosine function.) We have

$$A^n = Ie^{i\tilde{\omega}\Delta t\, n} = Ie^{i\tilde{\omega}t} = I\cos(\tilde{\omega}t) + iI\sin(\tilde{\omega}t)\,.$$

he physically relevant numerical solution can be taken as the real part of this omplex expression.

The calculations goes as

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

he last line follows from the relation $\cos x - 1 = -2\sin^2(x/2)$ (try `cos(x)-1` ı wolframalpha.com[5] to see the formula).

The scheme (7) with $u^n = Ie^{i\omega\tilde{\Delta}t\, n}$ inserted now gives

$$-Ie^{i\tilde{\omega}t}\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2}) + \omega^2 Ie^{i\tilde{\omega}t} = 0, \tag{15}$$

hich after dividing by $Ie^{i\tilde{\omega}t}$ results in

$$\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2}) = \omega^2\,. \tag{16}$$

he first step in solving for the unknown $\tilde{\omega}$ is

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

[5] `http://www.wolframalpha.com`

Then, taking the square root, applying the inverse sine function, and mul by $2/\Delta t$, results in

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

The first observation of (17) tells that there is a phase error si numerical frequency $\tilde{\omega}$ never equals the exact frequency ω. But how the approximation (17)? That is, what is the error $\omega - \tilde{\omega}$ or $\tilde{\omega}/\omega$? Tayl expansion for small Δt may give an expression that is easier to understa the complicated function in (17):

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

This means that

$$\tilde{\omega} = \omega\left(1 + \frac{1}{24}\omega^2\Delta t^2\right) + \mathcal{O}(\Delta t^4)\,.$$

The error in the numerical frequency is of second-order in Δt, and tl vanishes as $\Delta t \to 0$. We see that $\tilde{\omega} > \omega$ since the term $\omega^3\Delta t^2/24 > 0$ a is by far the biggest term in the series expansion for small $\omega\Delta t$. A nu frequency that is too large gives an oscillating curve that oscillates too f therefore "lags behind" the exact oscillations, a feature that can be see plots.

Figure 2 plots the discrete frequency (17) and its approximation $\omega = 1$ (based on the program `vib_plot_freq.py`[6]). Although $\tilde{\omega}$ is a f of Δt in (18), it is misleading to think of Δt as the important discre parameter. It is the product $\omega\Delta t$ that is the key discretization paramet quantity reflects the *number of time steps per period* of the oscillations this, we set $P = N_P\Delta t$, where P is the length of a period, and N_P is the of time steps during a period. Since P and ω are related by $P = 2\pi/\omega$ that $\omega\Delta t = 2\pi/N_P$, which shows that $\omega\Delta t$ is directly related to N_P.

The plot shows that at least $N_P \sim 25 - 30$ points per period are n for reasonable accuracy, but this depends on the length of the simulatio the total phase error due to the frequency error grows linearly with ti Exercise 2).

4.2 Exact discrete solution

Perhaps more important than the $\tilde{\omega} = \omega + \mathcal{O}(\Delta t^2)$ result found above is that we have an exact discrete solution of the problem:

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

[6] `http://tinyurl.com/jvzzcfn/vib/vib_plot_freq.py`

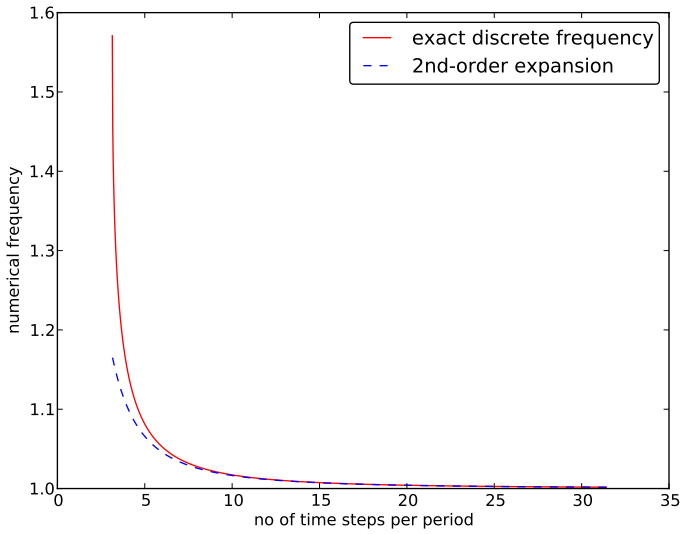


Figure 2: Exact discrete frequency and its second-order series expansion.

/e can then compute the error mesh function

$$e^n = u_{\mathrm{e}}(t_n) - u^n = I\cos(\omega n\Delta t) - I\cos(\tilde{\omega} n\Delta t)\ . \tag{20}$$

ı particular, we can use this expression to show *convergence* of the numerical :heme, i.e., $e^n \to 0$ as $\Delta t \to 0$. We have that

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

y L'Hopital's rule or simply asking `(2/x)*asin(w*x/2) as x->0` in Wolfra-ıAlpha[7]. Therefore, $\tilde{\omega}\to\omega$, and the two terms in e^n cancel each other in the mit $\Delta t\to 0$.

The error mesh function is ideal for verification purposes (and you are ıcouraged to make a test based on (19) in Exercise 10).

.3 The global error

o achieve more analytical insight into the nature of the global error, we can aylor expand the error mesh function. Since $\tilde{\omega}$ contains Δt in the denominator e use the series expansion for $\tilde{\omega}$ inside the cosine function:

[7] `http://www.wolframalpha.com/input/?i=%282%2Fx%29*asin%28w*x%2F2%29+as+x-%3E0`


```
>>> dt, w, t = symbols('dt w t')
>>> w_tilde_e = 2/dt*asin(w*dt/2)
>>> w_tilde_series = w_tilde_e.series(dt, 0, 4)
>>> # Get rid of O() term
>>> w_tilde_series = sum(w_tilde_series.as_ordered_terms()[:-1])
>>> w_tilde_series
dt**2*w**3/24 + w
>>> error = cos(w*t) - cos(w_tilde_series*t)
>>> error.series(dt, 0, 6)
dt**2*t*w**3*sin(t*w)/24 + dt**4*t**2*w**6*cos(t*w)/1152 + O(dt**
>>> error.series(dt, 0, 6).as_leading_term(dt)
dt**2*t*w**3*sin(t*w)/24
```

This means that the leading order global (true) error at a point t is prop to $\omega^3 t\Delta t^2$. Setting $t = n\Delta t$ and replacing $\sin(\omega t)$ by its maximum valı have the analytical leading-order expression

$$e^n = \frac{1}{24}n\omega^3\Delta t^3,$$

and the ℓ^2 norm of this error can be computed as

$$||e^n||^2_{\ell^2} = \Delta t\sum_{n=0}^{N_t}\frac{1}{24^2}n^2\omega^6\Delta t^6 = \frac{1}{24^2}\omega^6\Delta t^7\sum_{n=0}^{N_t}n^2\,.$$

The sum $\sum_{n=0}^{N_t}n^2$ is approximately equal to $\frac{1}{3}N_t^3$. Replacing N_t by $T/$ taking the square root gives the expression

$$||e^n||_{\ell^2} = \frac{1}{24}\sqrt{\frac{T^3}{3}}\omega^3\Delta t^2,$$

which shows that also the integrated error is proportional to Δt^2.

4.4 Stability

Looking at (19), it appears that the numerical solution has constant and amplitude, but an error in the frequency (phase error). However, there is error that is more serious, namely an unstable growing amplitude that ca of Δt is too large.

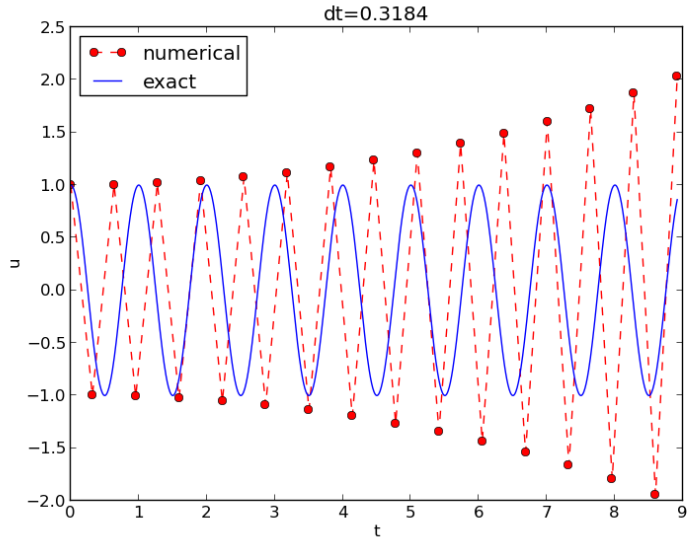
We realize that a constant amplitude demands $\tilde{\omega}$ to be a real nun complex $\tilde{\omega}$ is indeed possible if the argument x of $\sin^{-1}(x)$ has magnituc than unity: $|x| > 1$ (type `asin(x)` in wolframalpha.com[8] to see basic prop $\sin^{-1}(x)$). A complex $\tilde{\omega}$ can be written $\tilde{\omega} = \tilde{\omega}_r + i\tilde{\omega}_i$. Since $\sin^{-1}(x)$ has a imaginary part for $x > 1$, $\tilde{\omega}_i < 0$, it means that $\exp(i\omega\tilde{t}) = \exp(-\tilde{\omega}_i t)$ ex will lead to exponential growth in time because $\exp(-\tilde{\omega}_i t)$ with $\tilde{\omega}_i <$ positive exponent.

We do not tolerate growth in the amplitude and we therefore have a *criterion* arising from requiring the argument $\omega\Delta t/2$ in the inverse sine to be less than one:

$$\frac{\omega\Delta t}{2} \le 1 \quad\Rightarrow\quad \Delta t \le \frac{2}{\omega}\,.$$

[8] `http://www.wolframalpha.com`

/ith $\omega = 2\pi$, $\Delta t > \pi^{-1} = 0.3183098861837907$ will give growing solutions. igure 3 displays what happens when $\Delta t = 0.3184$, which is slightly above the :itical value: $\Delta t = \pi^{-1} + 9.01 \cdot 10^{-5}$.



igure 3: Growing, unstable solution because of a time step slightly beyond ıe stability limit.

.5 About the accuracy at the stability limit

n interesting question is whether the stability condition $\Delta t < 2/\omega$ is unfortunate, r more precisely: would it be meaningful to take larger time steps to speed p computations? The answer is a clear no. At the stability limit, we have ıat $\sin^{-1} \omega\Delta t/2 = \sin^{-1} 1 = \pi/2$, and therefore $\tilde{\omega} = \pi/\Delta t$. (Note that the pproximate formula (18) is very inaccurate for this value of Δt as it predicts $= 2.34/pi$, which is a 25 percent reduction.) The corresponding period of the umerical solution is $\tilde{P} = 2\pi/\tilde{\omega} = 2\Delta t$, which means that there is just one time :ep Δt between a peak and a through in the numerical solution. This is the ıortest possible wave that can be represented in the mesh. In other words, it is ot meaningful to use a larger time step than the stability limit.

Also, the phase error when $\Delta t = 2/\omega$ is severe: Figure 4 shows a comparison f the numerical and analytical solution with $\omega = 2\pi$ and $\Delta t = 2/\omega = \pi^{-1}$. lready after one period, the numerical solution has a through while the exact)lution has a peak (!). The error in frequency when Δt is at the stability limit ecomes $\omega - \tilde{\omega} = \omega(1 - \pi/2) \approx -0.57\omega$. The corresponding error in the period $P - \tilde{P} \approx 0.36P$. The error after m periods is then $0.36mP$. This error has :ach half a period when $m = 1/(2 \cdot 0.36) \approx 1.38$, which theoretically confirms

the observations in Figure 4 that the numerical solution is a through ah peak already after one and a half period.

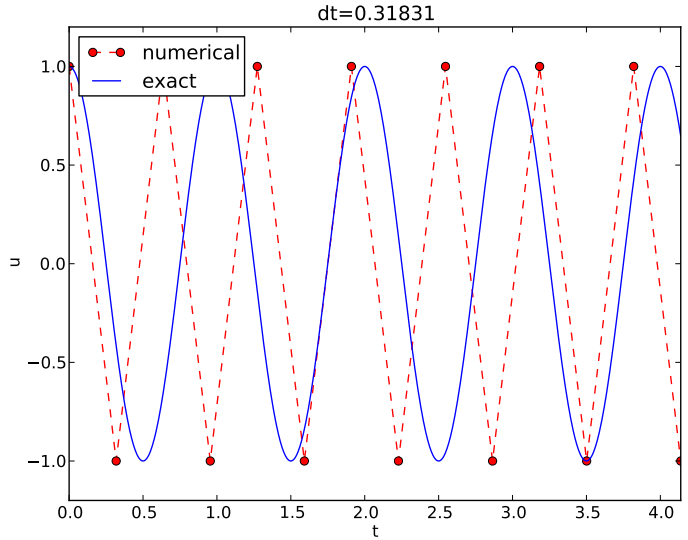


Figure 4: Numerical solution with Δt exactly at the stability limi

Summary.

From the accuracy and stability analysis we can draw three impo conclusions:

1. The key parameter in the formulas is $p = \omega\Delta t$. The peri oscillations is $P = 2\pi/\omega$, and the number of time steps per per $N_P = P/\Delta t$. Therefore, $p = \omega\Delta t = 2\pi N_P$, showing that the cı parameter is the number of time steps per period. The sm possible N_P is 2, showing that $p \in (0, \pi]$.

2. Provided $p \leq 2$, the amplitude of the numerical solution is cons

3. The numerical solution exhibits a relative phase error $\tilde{\omega}/\omega \approx 1 +$ This error leads to wrongly displaced peaks of the numerical sol and the error in peak location grows linearly with time (see Exerc

Alternative schemes based on 1st-order equations

standard technique for solving second-order ODEs is to rewrite them as a ystem of first-order ODEs and then apply the vast collection of methods for rst-order ODE systems. Given the second-order ODE problem

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

e introduce the auxiliary variable $v = u'$ and express the ODE problem in erms of first-order derivatives of u and v:

$$u' = v, \tag{22}$$
$$v' = -\omega^2 u\,. \tag{23}$$

he initial conditions become $u(0) = I$ and $v(0) = 0$.

.1 Standard methods for 1st-order ODE systems

he Forward Euler scheme. A Forward Euler approximation to our 2×2 ystem of ODEs (22)-(23) becomes

$$[D_t^+ u = v]^n, [D_t^+ v = -\omega^2 u]^n, \tag{24}$$

r written out,

$$u^{n+1} = u^n + \Delta t v^n, \tag{25}$$
$$v^{n+1} = v^n - \Delta t \omega^2 u^n\,. \tag{26}$$

Let us briefly compare this Forward Euler method with the centered difference cheme for the second-order differential equation. We have from (25) and (26) pplied at levels n and $n-1$ that

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

ince from (25)

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

follows that

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

hich is very close to the centered difference scheme, but the last term is evaluated t t_{n-1} instead of t_n. This difference is actually crucial for the accuracy of the orward Euler method applied to vibration problems.

The Backward Euler scheme. A Backward Euler approximation t system is equally easy to write up in the operator notation:

$$[D_t^- u = v]^{n+1},$$
$$[D_t^- v = -\omega u]^{n+1}\,.$$

This becomes a coupled system for u^{n+1} and v^{n+1}:

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

The Crank-Nicolson scheme. The Crank-Nicolson scheme takes t in the operator notation:

$$[D_t u = \overline{v}^t]^{n+\frac{1}{2}},$$
$$[D_t v = -\omega \overline{u}^t]^{n+\frac{1}{2}}\,.$$

Writing the equations out shows that is also a coupled system:

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

Comparison of schemes. We can easily compare methods like the on (and many more!) with the aid of the Odespy[9] package. Below is a sketc code.

```
import odespy
import numpy as np

def f(u, t, w=1):
    u, v = u  # u is array of length 2 holding our [u, v]
    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  # 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([I, 0])
        u, t = solver.solve(t_mesh)
```

[9] https://github.com/hplgit/odespy

here is quite some more code dealing with plots also, and we refer to the source le `vib_undamped_odespy.py`[10] for details. Observe that keyword arguments in `(u,t,w=1)` can be supplied through a solver parameter `f_kwargs` (dictionary).

Specification of the Forward Euler, Backward Euler, and Crank-Nicolson chemes is done like this:

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

The `vib_undamped_odespy.py` program makes two plots of the computed olutions with the various methods in the `solvers` list: one plot with $u(t)$ versus and one *phase plane plot* where v is plotted against u. That is, the phase lane plot is the curve $(u(t), v(t))$ parameterized by t. Analytically, $u = I\cos(\omega t)$ nd $v = u' = -\omega I\sin(\omega t)$. The exact curve $(u(t), v(t))$ is therefore an ellipse, hich often looks like a circle in a plot if the axes are automatically scaled. he important feature, however, is that exact curve $(u(t), v(t))$ is closed and epeats itself for every period. Not all numerical schemes are capable to do that, eaning that the amplitude instead shrinks or grows with time.

The Forward Euler scheme in Figure 5 has a pronounced spiral curve, pointing o the fact that the amplitude steadily grows, which is also evident in Figure 6. he Backward Euler scheme has a similar feature, except that the spriral goes ward and the amplitude is significantly damped. The changing amplitude nd the sprial form decreases with decreasing time step. The Crank-Nicolson cheme looks much more accurate. In fact, these plots tell that the Forward and ackward Euler schemes are not suitable for solving our ODEs with oscillating olutions.

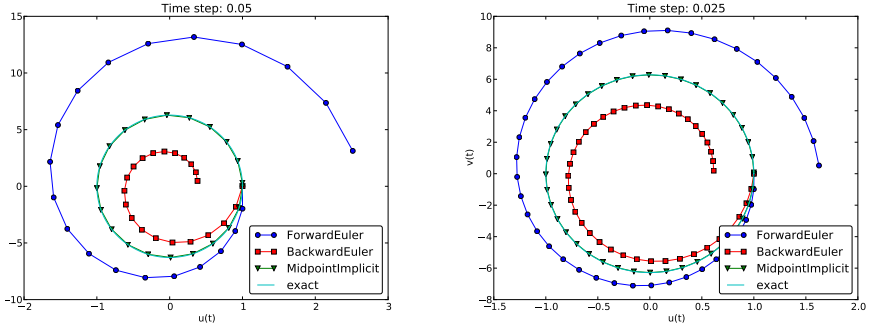


Figure 5: Comparison of classical schemes in the phase plane.

We may run two popular standard methods for first-order ODEs, the 2nd- nd 4th-order Runge-Kutta methods, to see how they perform. Figures 7 and 8

[10] `http://tinyurl.com/jvzzcfn/vib/vib_undamped_odespy.py`

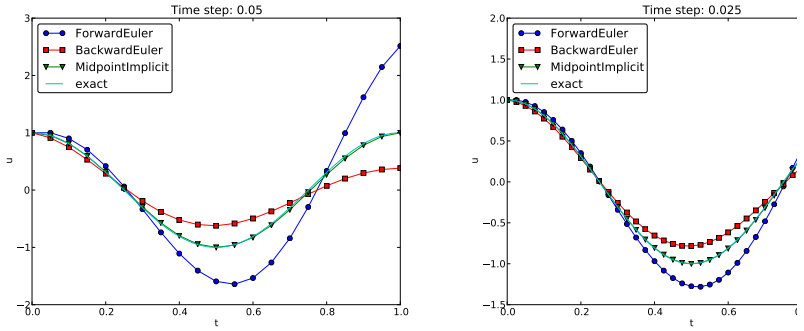


Figure 6: Comparison of classical schemes.

show the solutions with larger Δt values than what was used in the p two plots.

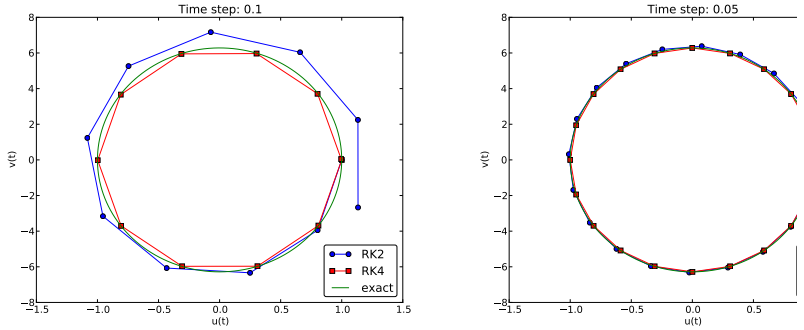


Figure 7: Comparison of Runge-Kutta schemes in the phase plan

The visual impression is that the 4th-order Runge-Kutta method accurate, under all circumstances in these tests, and the 2nd-order schen from amplitude errors unless the time step is very small.

The corresponding results for the Crank-Nicolson scheme are sh Figures 9 and 10. It is clear that the Crank-Nicolson scheme outp the 2nd-order Runge-Kutta method. Both schemes have the same o accuracy $\mathcal{O}(\Delta t^2)$, but their differences in the accuracy that matters i physical application is very clearly pronounced in this example. Exe invites you to investigate how

5.2 Enegy considerations

The observations of various methods in the previous section can be interpreted if we compute an quantity reflecting the total *energy of the* It turns out that this quantity,

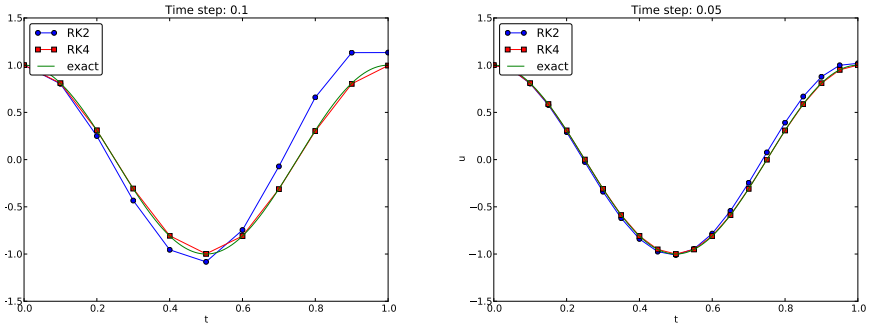
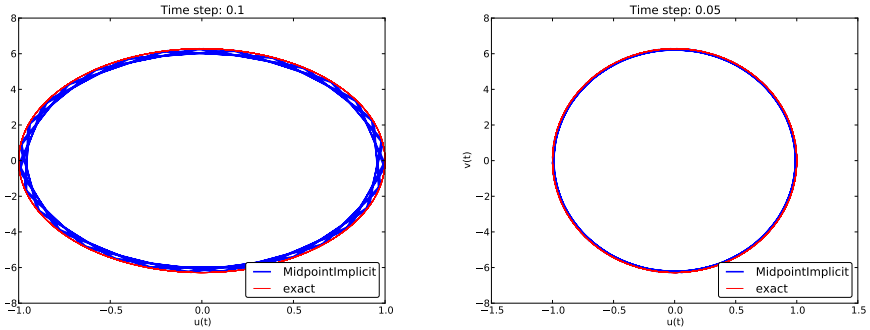


Figure 8: Comparison of Runge-Kutta schemes.



igure 9: Long-time behavior of the Crank-Nicolson scheme in the phase plane.

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

constant for all t. Checking that $E(t)$ really remains constant brings evidence hat the numerical computations are sound. Such energy measures, when they xist, are much used to check numerical simulations.

)erivation of the energy expression. We starting multiplying

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

y u' and integrating from 0 to T:

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

)bserving that

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

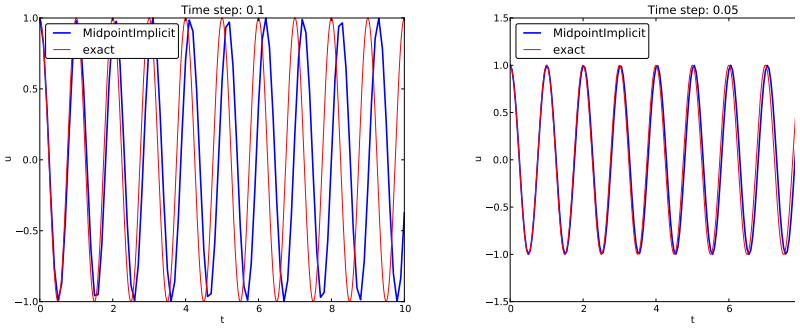


Figure 10: Long-time behavior of the Crank-Nicolson scheme.

we get

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

where we have introduced the energy measure $E(t)$

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

The important result from this derivation is that the total energy is co

$$E(t) = \text{const}\,.$$

Remark on the energy expression.

The quantity $E(t)$ derived above is physically not the energy of a vib mechanical system, but the energy per unit mass. To see this, we start Newton's second law $F = ma$ (F is the sum of forces, m is the m the system, and a is the acceleration). The displacement u is relate through $a = u''$. With a spring force as the only force we have $F =$ where k is a spring constant measuring the stiffness of the spring. Ne second law then implies the differential equation

$$-ku = mu'' \quad \Rightarrow mu'' + ku = 0\,.$$

This equation of motion can be turned into an energy balance equati finding the work done by each term during a time interval $[0, T]$. T end, we multiply the equation by $du = u'dt$ and integrate:

$$\int_0^T muu'dt + \int_0^T kuu'dt = 0\,.$$

The result is

$$E(t) = E_k(t) + E_p(t) = 0,$$

where

$$E_k(t) = \frac{1}{2}mv^2, \quad v = u', \tag{36}$$

is the *kinetic energy* of the system,

$$E_p(t) = \frac{1}{2}ku^2 \tag{37}$$

is the *potential energy*, and the sum $E(t)$ is the total energy. The derivation demonstrates the famous energy principle that any change in the kinetic energy is due to a change in potential energy and vice versa.

The equation $mu'' + ku = 0$ can be divided by m and written as $u'' + \omega^2 u = 0$ for $\omega = \sqrt{k/m}$. The energy expression $E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2 u^2$ derived earlier is then simply the true physical total energy $\frac{1}{2}m(u')^2 + \frac{1}{2}k^2u^2$ divided by m, i.e., total energy per unit mass.

Example. Analytically, we have $u(t) = I\cos\omega t$, if $u(0) = I$ and $u'(0) = 0$, so we can easily check that the evolution of the energy $E(t)$ is constant:

$$E(t) = \frac{1}{2}I^2(-\omega\sin\omega t)^2 + \frac{1}{2}\omega^2 I^2\cos^2\omega t = \frac{1}{2}\omega^2(\sin^2\omega t + \cos^2\omega t) = \frac{1}{2}\omega^2\,.$$

Discrete total energy. The total energy $E(t)$ can be computed as soon as u^n is available. Using $(u')^n \approx [D_{2t}u^n]$ we have

$$E^n = \frac{1}{2}([D_{2t}u]^n)^2 + \frac{1}{2}\omega^2(u^n)^2\,.$$

The errors involved in E^n get a contribution $\mathcal{O}(\Delta t^2)$ from the difference approximation of u' and a contribution from the numerical error in u^n. With a second-order scheme for computing u^n, the overall error in E^n is expected to be $\mathcal{O}(\Delta t^2)$.

An error measure based on total energy. The error in total energy, as a mesh function, can be computed by

$$e_E^n = \frac{1}{2}\left(\frac{u^{n+1} - u^{n-1}}{2\Delta t}\right)^2 + \frac{1}{2}\omega^2(u^n)^2 - E(0), \quad n = 1, \ldots, N_t - 1, \tag{38}$$

where

$$E(0) = \frac{1}{2}V^2 + \frac{1}{2}\omega^2 I^2,$$

$u(0) = I$ and $u'(0) = V$. A useful norm can be the maximum absolute value of e_E^n:

$$||e_E^n||_{\ell^\infty} = \max_{1\leq n<N_t} |e_E^n|\,.$$

The corresponding Python implementation takes the form

```
# import numpy as np and compute u, t
dt = t[1]-t[0]
E = 0.5*((u[2:] - u[:-2])/(2*dt))**2 + 0.5*w**2*u[1:-1]**2
E0 = 0.5*V**2 + 0.5**w**2*I**2
e_E = E - E0
e_E_norm = np.abs(e_E).max()
```

The convergence rates of the quantity `e_E_norm` can be used for veri The value of `e_E_norm` is also useful for comparing schemes through thei to preserve energy. Below is a table demonstrating the error in total for various schemes. We clearly see that the Crank-Nicolson and 4t Runge-Kutta schemes are superior to the 2nd-order Runge-Kutta metl even more superior to the Forward and Backward Euler schemes.

Method	T	Δt	$\max\|e_E^n\|$
Forward Euler	1	0.05	$1.113 \cdot 10^2$
Forward Euler	1	0.025	$3.312 \cdot 10^1$
Backward Euler	1	0.05	$1.683 \cdot 10^1$
Backward Euler	1	0.025	$1.231 \cdot 10^1$
Runge-Kutta 2nd-order	1	0.1	8.401
Runge-Kutta 2nd-order	1	0.05	$9.637 \cdot 10^{-1}$
Crank-Nicolson	1	0.05	$9.389 \cdot 10^{-1}$
Crank-Nicolson	1	0.025	$2.411 \cdot 10^{-1}$
Runge-Kutta 4th-order	1	0.1	2.387
Runge-Kutta 4th-order	1	0.05	$6.476 \cdot 10^{-1}$
Crank-Nicolson	10	0.1	3.389
Crank-Nicolson	10	0.05	$9.389 \cdot 10^{-1}$
Runge-Kutta 4th-order	10	0.1	3.686
Runge-Kutta 4th-order	10	0.05	$6.928 \cdot 10^{-1}$

5.3 The Euler-Cromer method

While the 4th-order Runge-Kutta method and the a centered Crank-N scheme work well for the first-order formulation of the vibration model, bc inferior to the straightforward centered difference scheme for the secor equation $u'' + \omega^2 u = 0$. However, there is a similarly successful scheme a for the first-order system $u' = v$, $v' = -\omega^2 u$, to be presented next.

Forward-backward discretization. The idea is to apply a Forwar discretization to the first equation and a Backward Euler discretizatio second. In operator notation this is stated as

$$[D_t^+ u = v]^n,$$
$$[D_t^- v = -\omega u]^{n+1}\,.$$

/e can write out the formulas and collect the unknowns on the left-hand side:

$$u^{n+1} = u^n + \Delta t v^n, \tag{41}$$
$$v^{n+1} = v^n - \Delta t \omega^2 u^{n+1}\,. \tag{42}$$

/e realize that u^{n+1} can be computed from (41) and then v^{n+1} from (42) using ıe recently computed value u^{n+1} on the right-hand side.

The scheme (41)-(42) goes under several names: Forward-backward scheme, emi-implicit Euler method[11], symplectic Euler, semi-explicit Euler, Newton-törmer-Verlet, and Euler-Cromer. We shall stick to the latter name. Since both me discretizations are based on first-order difference approximation, one may ıink that the scheme is only of first-order, but this is not true: the use of a ɔrward and then a backward difference make errors cancel so that the overall :ror in the scheme is $\mathcal{O}(\Delta t^2)$. This is explaned below.

quivalence with the scheme for the second-order ODE. We may elim-ıate the v^n variable from (41)-(42). From (42) we have $v^n = v^{n-1} - \Delta t\omega^2 u^n$, hich can be inserted in (41) to yield

$$u^{n+1} = u^n + \Delta t v^{n-1} - \Delta t^2\omega^2 u^n. \tag{43}$$

he v^{n-1} quantity can be expressed by u^n and u^{n-1} using (41):

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

nd when this is inserted in (43) we get

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2\omega^2 u^n, \tag{44}$$

hich is nothing but the centered scheme (7)! The previous analysis of this :heme then also applies to the Euler-Cromer method. That is, the amplitude constant, given that the stability criterion is fulfilled, but there is always a hase error (18).

The initial condition $u' = 0$ means $u' = v = 0$. Then $v^0 = 0$, and (41) implies $^1 = u^0$, while (42) says $v^1 = -\omega^2 u^0$. This approximation, $u^1 = u^0$, corresponds ɔ a first-order Forward Euler discretization of the initial condition $u'(0) = 0$: $\mathcal{D}_t^+ u = 0]^0$. Therefore, the Euler-Cromer scheme will start out differently and ot exactly reproduce the solution of (7).

.4 The Euler-Cromer scheme on a staggered mesh

he Forward and Backward Euler schemes used in the Euler-Cromer method are oth non-symmetric, but their combination yields a symmetric method since the :sulting scheme is equivalent with a centered (symmetric) difference scheme for $'' + \omega^2 u = 0$. The symmetric nature of the Euler-Cromer scheme is much more vident if we introduce a *staggered mesh* in time where u is sought at integer

[11] http://en.wikipedia.org/wiki/Semi-implicit_Euler_method

time points t_n and v is sought at $t_{n+1/2}$ *between* two u points. The unkn then $u^1, v^{3/2}, u^2, v^{5/2}$, and so on. We typically use the notation u^n and for the two unknown mesh functions.

On a staggered mesh it is natural to use centered difference approxi expressed in operator notation as

$$[D_t u = v]^{n+\frac{1}{2}},$$
$$[D_t v = -\omega u]^{n+1}\,.$$

Writing out the formulas gives

$$u^{n+1} = u^n + \Delta t v^{n+\frac{1}{2}},$$
$$v^{n+\frac{3}{2}} = v^{n+\frac{1}{2}} - \Delta t\omega^2 u^{n+1}\,.$$

Of esthetic reasons one often writes these equations at the previous time replace the $\frac{3}{2}$ by $\frac{1}{2}$ in the left-most term in the last equation,

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

Such a rewrite is only mathematical cosmetics. The important thing is t centered scheme has exactly the same computational structure as the f backward scheme. We shall use the names *forward-backward Euler-Cro staggered Euler-Cromer* to distinguish the two schemes.

We can eliminate the v values and get back the centered scheme b the second-order differential equation, so all these three schemes are eq However, they differ somewhat in the treatment of the initial condition

Suppose we have $u(0) = I$ and $u'(0) = v(0) = 0$ as mathematica conditions. This means $u^0 = I$ and

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

Using the discretized equation (50) for $n = 0$ yields

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

and eliminating $v^{-\frac{1}{2}} = -v^{\frac{1}{2}}$ results in $v^{\frac{1}{2}} = -\frac{1}{2}\Delta t\omega^2 I$ and

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

which is exactly the same equation for u^1 as we had in the centered schem on the second-order differential equation (and hence corresponds to a c difference approximation of the initial condition for $u'(0)$). The conclusio a staggered mesh is fully equivalent with that scheme, while the forward-b version gives a slight deviation in the computation of u^1.

We can redo the derivation of the initial conditions when $u'(0) = V$:

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

sing this $v^{-\frac{1}{2}}$ in

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

ıen gives $v^{\frac{1}{2}} = V - \frac{1}{2}\Delta t\omega^2 I$. The general initial conditions are therefore

$$u^0 = I, \tag{51}$$

$$v^{\frac{1}{2}} = V - \frac{1}{2}\Delta t\omega^2 I\,. \tag{52}$$

.5 Implementation of the scheme on a staggered mesh

he algorithm goes like this:

1. Set the initial values (51) and (52).
2. For $n = 1, 2, \ldots$:
   (a) Compute u^n from (49).
   (b) Compute $v^{n+1/2}$ from (50).

mplementation with integer indices. Translating the schemes (49) and 50) to computer code faces the problem of how to store and access $v^{n+\frac{1}{2}}$, nce arrays only allow integer indices with base 0. We must then introduce a onvention: $v^{1+\frac{1}{2}}$ is stored in `v[n]` while $v^{1-\frac{1}{2}}$ is stored in `v[n-1]`. We can then rite the algorithm in Python as

```
def solver(I, w, dt, T):
    dt = float(dt)
    Nt = int(round(T/dt))
    u = zeros(Nt+1)
    v = zeros(Nt+1)
    t = linspace(0, Nt*dt, Nt+1)  # mesh for u
    t_v = t + dt/2                # mesh for v

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

ote that the return u and v together with the mesh points such that the omplete mesh function for u is described by `u` and `t`, while `v` and `t_v` represents ıe mesh function for v.

Implementation with half-integer indices. Some prefer to see relationship between the code and the mathematics for the quantiti half-integer indices. For example, we would like to replace the updating e for `v[n]` by

```
v[n+half] = v[n-half] - dt*w**2*u[n]
```

This is easy to do if we could be sure that `n+half` means `n` and `n-hal` `n-1`. A possible solution is to define `half` as a special object such that ar plus `half` results in the integer, while an integer minus `half` equals the minus 1. A simple Python class may realize the `half` object:

```
class HalfInt:
    def __radd__(self, other):
        return other

    def __rsub__(self, other):
        return other - 1

half = HalfInt()
```

The `__radd__` function is invoked for all expressions `n+half` ("right ad `self` as `half` and `other` as `n`). Similarly, the `__rsub__` function is inv `n-half` and results in `n-1`.

Using the `half` object, we can implement the algorithms in an eve readable way:

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

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

Verification of this code is easy as we can just compare the com with the `u` produced by the `solver` function in `vib_undamped.py` (whic $u'' + \omega^2 u = 0$ directly). The values should coincide to machine precisi the two numerical methods are mathematically equivalent. We refer to `vib_undamped_staggered.py`[12] for the details of a nose test that che property.

[12] `http://tinyurl.com/jvzzcfn/vib/vib_undamped_staggered.py`

5 Generalization: damping, nonlinear spring, and external excitation

We shall now generalize the simple model problem from Section 1 to include a possibly nonlinear damping term $f(u')$, a possibly nonlinear spring (or restoring) force $s(u)$, and some external excitation $F(t)$:

$$mu'' + f(u') + s(u) = F(t), \quad u(0) = I,\ u'(0) = V,\ t \in (0,T]\,. \tag{53}$$

We have also included a possibly nonzero initial value of $u'(0)$. The parameters m, $f(u')$, $s(u)$, $F(t)$, I, V, and T are input data.

There are two main types of damping (friction) forces: linear $f(u') = bu$, or quadratic $f(u') = bu'|u'|$. Spring systems often feature linear damping, while air resistance usually gives rise to quadratic damping. Spring forces are often linear: $s(u) = cu$, but nonlinear versions are also common, the most famous is the gravity force on a pendulum that acts as a spring with $s(u) \sim \sin(u)$.

5.1 A centered scheme for linear damping

Sampling (53) at a mesh point t_n, replacing $u''(t_n)$ by $[D_tD_tu]^n$, and $u'(t_n)$ by $[D_{2t}u]^n$ results in the discretization

$$[mD_tD_tu + f(D_{2t}u) + s(u) = F]^n, \tag{54}$$

which written out means

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + f(\frac{u^{n+1} - u^{n-1}}{2\Delta t}) + s(u^n) = F^n, \tag{55}$$

where F^n as usual means $F(t)$ evaluated at $t = t_n$. Solving (55) with respect to the unknown u^{n+1} gives a problem: the u^{n+1} inside the f function makes the equation *nonlinear* unless $f(u')$ is a linear function, $f(u') = bu'$. For now we shall assume that f is linear in u'. Then

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + b\frac{u^{n+1} - u^{n-1}}{2\Delta t} + s(u^n) = F^n, \tag{56}$$

which gives an explicit formula for u at each new time level:

$$u^{n+1} = (2mu^n + (\frac{b}{2}\Delta t - m)u^{n-1} + \Delta t^2(F^n - s(u^n)))(m + \frac{b}{2}\Delta t)^{-1}\,. \tag{57}$$

For the first time step we need to discretize $u'(0) = V$ as $[D_{2t}u = V]^0$ and combine with (57) for $n = 0$. The discretized initial condition leads to

$$u^{-1} = u^1 - 2\Delta t V, \tag{58}$$

which inserted in (57) for $n = 0$ gives an equation that can be solved for u^1:

$$u^1 = u^0 + \Delta t\, V + \frac{\Delta t^2}{2m}(-bV - s(u^0) + F^0)\,. \tag{59}$$

6.2 A centered scheme for quadratic damping

When $f(u') = bu'|u'|$, we get a quadratic equation for u^{n+1} in (55) equation can straightforwardly be solved, but we can also avoid the nonl by performing an approximation that is within other numerical errors have already committed by replacing derivatives with finite differences.

The idea is to reconsider (53) and only replace u'' by D_tD_tu, giving

$$[mD_tD_tu + bu'|u'| + s(u) = F]^n,$$

Here, $u'|u'|$ is to be computed at time t_n. We can introduce a *geometri* defined by

$$(w^2)^n \approx w^{n-1/2}w^{n+1/2},$$

for some quantity w depending on time. The error in the geometri approximation is $\mathcal{O}(\Delta t^2)$, the same as in the approximation $u'' \approx D_tD_t$ $w = u'$ it follows that

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

The next step is to approximate u' at $t_{n\pm1/2}$, but here a centered differe be used:

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

We then get

$$[u'|u'|]^n \approx [D_tu]^{n+1/2}|[D_tu]^{n-1/2}| = \frac{u^{n+1} - u^n}{\Delta t}\frac{|u^n - u^{n-1}|}{\Delta t}\,.$$

The counterpart to (55) is then

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

which is linear in u^{n+1}. Therefore, we can easily solve with respect to u achieve the explicit updating formula

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

In the derivation of a special equation for the first time step we h some trouble: inserting (58) in (64) for $n = 0$ results in a complicated n equation for u^1. By thinking differently about the problem we can ea away with the nonlinearity again. We have for $n = 0$ that $b[u'|u'|]^0 =$ Using this value in (60) gives

$$[mD_tD_tu + bV|V| + s(u) = F]^0 \,. \tag{65}$$

/riting this equation out and using (58) results in the special equation for the rst time step:

$$u^1 = u^0 + \Delta t V + \frac{\Delta t^2}{2m}\left(-bV|V| - s(u^0) + F^0\right) \,. \tag{66}$$

.3 A forward-backward discretization of the quadratic damping term

he previous section first proposed to discretize the quadratic damping term $\iota'|u'$ using centered differences: $[|D_{2t}|D_{2t}u]^n$. As this gives rise to a nonlinearity ı u^{n+1}, it was instead proposed to use a geometric mean combined with centered ifferences. But there are other alternatives. To get rid of the nonlinearity in $D_{2t}|D_{2t}u]^n$, one can think differently: apply a backward difference to $|u'|$, such ıat the term involves known values, and apply a forward difference to u' to ıake the term linear in the unknown u^{n+1}. With mathematics,

$$[\beta|u'|u']^n \approx \beta|[D_t^- u]^n|[D_t^+ u]^n = \beta\left|\frac{u^- u^{n-1}}{\Delta t}\right|\frac{u^{n+1}-u^n}{\Delta t} \,.$$

he forward and backward differences have both an error proportional to Δt so ne may think the discretization above leads to a first-order scheme. However, by ɔoking at the formulas, we realize that the forward-backward differences result ı exactly the same scheme as when we used a geometric mean and centered ifferences. Therefore, the forward-backward differences act in a symmetric way nd actually produce a second-order accurate discretization of the quadratic amping term.

.4 Implementation

he algorithm arising from the methods in Sections 6.1 and 6.2 is very similar ɔ the undamped case in Section 1.2. The difference is basically a question f different formulas for u^1 and u^{n+1}. This is actually quite remarkable. The quation (53) is normally impossible to solve by pen and paper, but possible ɔr some special choices of F, s, and f. On the contrary, the complexity of the onlinear generalized model (53) versus the simple undamped model is not a big eal when we solve the problem numerically!

The computational algorithm takes the form

1. $u^0 = I$
2. compute u^1 from (59) if linear damping or (66) if quadratic damping
3. for $n = 1, 2, \ldots, N_t - 1$:
   (a) compute u^{n+1} from (57) if linear damping or (64) if quadratic damping

Modifying the `solver` function for the undamped case is fairly easy, difference being many more terms and if tests on the type of damping:

```
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 di
    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
                      + dt**2*(F(t[n]) - s(u[n])))/\
                      (m + b*abs(u[n] - u[n-1]))
    return u, t
```

The complete code resides in the file `vib.py`[13].

6.5 Verification

Constant solution. For debugging and initial verification, a constant is often very useful. We choose $u_e(t) = I$, which implies $V = 0$. Inserte ODE, we get $F(t) = s(I)$ for any choice of f. Since the discrete derivat constant vanishes (in particular, $[D_{2t}I]^n = 0$, $[D_tI]^n = 0$, and $[D_tD_tI$ the constant solution also fulfills the discrete equations. The constant therefore be reproduced to machine precision.

Linear solution. Now we choose a linear solution: $u_e = ct + d$. Th condition $u(0) = I$ implies $d = I$, and $u'(0) = V$ forces c to be V. In $u_e = Vt + I$ in the ODE with linear damping results in

$$0 + bV + s(Vt + I) = F(t),$$

while quadratic damping requires the source term

$$0 + b|V|V + s(Vt + I) = F(t) \,.$$

[13] `http://tinyurl.com/jvzzcfn/vib/vib.py`

ince the finite difference approximations used to compute u' all are exact for a near function, it turns out that the linear u_e is also a solution of the discrete quations. Exercise 9 asks you to carry out all the details.

uadratic solution. Choosing $u_e = bt^2 + Vt + I$, with b arbitrary, fulfills ne initial conditions and fits the ODE if F is adjusted properly. The solution lso solves the discrete equations with linear damping. However, this quadratic olynomial in t does not fulfill the discrete equations in case of quadratic damping, ecause the geometric mean used in the approximation of this term introduces n error. Doing Exercise 9 will reveal the details. One can fit F^n in the discrete quations such that the quadratic polynomial is reproduced by the numerical ıethod (to machine precision).

.6 Visualization

he functions for visualizations differ significantly from those in the undamped ase in the `vib_undamped.py` program because we in the present general case do ot have an exact solution to include in the plots. Moreover, we have no good stimate of the periods of the oscillations as there will be one period determined y the system parameters, essentially the approximate frequency $\sqrt{s'(0)/m}$ or linear s and small damping, and one period dictated by $F(t)$ in case the xcitation is periodic. This is, however, nothing that the program can depend n or make use of. Therefore, the user has to specify T and the window width ı case of a plot that moves with the graph and shows the most recent parts of in long time simulations.

The `vib.py` code contains several functions for analyzing the time series gnal and for visualizing the solutions.

.7 User interface

he `main` function has substantial changes from the `vib_undamped.py` code nce we need to specify the new data c, $s(u)$, and $F(t)$. In addition, we must et T and the plot window width (instead of the number of periods we want to mulate as in `vib_undamped.py`). To figure out whether we can use one plot or the whole time series or if we should follow the most recent part of u, we an use the `plot_empricial_freq_and_amplitude` function's estimate of the umber of local maxima. This number is now returned from the function and sed in `main` to decide on the visualization technique.

```
def main():
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--I', type=float, default=1.0)
    parser.add_argument('--V', type=float, default=0.0)
    parser.add_argument('--m', type=float, default=1.0)
    parser.add_argument('--c', type=float, default=0.0)
    parser.add_argument('--s', type=str, default='u')
    parser.add_argument('--F', type=str, default='0')
    parser.add_argument('--dt', type=float, default=0.05)
    parser.add_argument('--T', type=float, default=140)
    parser.add_argument('--damping', type=str, default='linear')
    parser.add_argument('--window_width', type=float, default=30)
```



```
    parser.add_argument('--savefig', action='store_true')
    a = parser.parse_args()
    from scitools.std import StringFunction
    s = StringFunction(a.s, independent_variable='u')
    F = StringFunction(a.F, independent_variable='t')
    I, V, m, c, dt, T, window_width, savefig, damping = \
       a.I, a.V, a.m, a.c, a.dt, a.T, a.window_width, a.savefig,
       a.damping

    u, t = solver(I, V, m, c, s, F, dt, T)
    num_periods = empirical_freq_and_amplitude(u, t)
    if num_periods <= 15:
        figure()
        visualize(u, t)
    else:
        visualize_front(u, t, window_width, savefig)
    show()
```

The program `vib.py` contains the above code snippets and can solve th problem (53). As a demo of `vib.py`, we consider the case $I = 1$, $V = 0$ $c = 0.03$, $s(u) = \sin(u)$, $F(t) = 3\cos(4t)$, $\Delta t = 0.05$, and $T = 140$. The command to run is

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

This results in a moving window following the function[14] on the screen. F shows a part of the time series.

6.8 A staggered Euler-Cromer scheme for the gener model

The model

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

can be rewritten as a first-order ODE system

$$u' = v,$$
$$v' = m^{-1}\left(F(t) - f(v) - s(u)\right).$$

It is natural to introduce a staggered mesh (see Section 5.4) and seek u points t_n (the numerical value is denoted by u^n) and v between mesl at $t_{n+1/2}$ (the numerical value is denoted by $v^{n+1/2}$). A centered di approximation to (68)-(69) can then be written in operator notation as

$$[D_t u = v]^{n-1/2},$$
$$[D_t v = m^{-1}\left(F(t) - f(v) - s(u)\right)]^n.$$

[14] `http://tinyurl.com/k3sdbuv/pub/mov-vib/vib_generalized_dt0.05/index.ht`

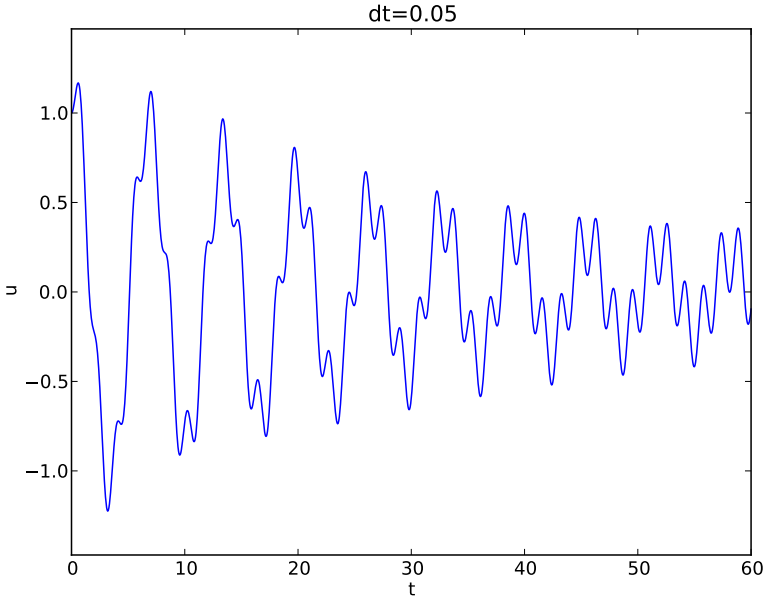


Figure 11: Damped oscillator excited by a sinusoidal function.

/ritten out,

$$\frac{u^n - u^{n-1}}{\Delta t} = v^{n-\frac{1}{2}}, \tag{72}$$

$$\frac{v^{n+\frac{1}{2}} - v^{n-\frac{1}{2}}}{\Delta t} = m^{-1}\left(F^n - f(v^n) - s(u^n)\right). \tag{73}$$

/ith linear damping, $f(v) = bv$, we can use an arithmetic mean for $f(v^n)$: $(v^n) \approx= \frac{1}{2}(f(v^{n-1/2}) + f(v^{n+1/2}))$. The system (72)-(73) can then be solved ith respect to the unknowns u^n and $v^{n+1/2}$:

$$u^n = u^{n-1} + \Delta t v^{n-\frac{1}{2}}, \tag{74}$$

$$v^{n+\frac{1}{2}} = \left(1 + \frac{b}{2m}\Delta t\right)^{-1}\left(v^{n-\frac{1}{2}} + \Delta t m^{-1}\left(F^n - \frac{1}{2}f(v^{n-\frac{1}{2}}) - s(u^n)\right)\right). \tag{75}$$

In case of quadratic damping, $f(v) = b|v|v$, we can use a geometric mean: $(v^n) \approx b|v^{n-1/2}|v^{n+1/2}$. Inserting this approximation in (72)-(73) and solving)r the unknowns u^n and $v^{n+1/2}$ results in

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

$$v^{n+\frac{1}{2}} = (1 + \frac{b}{m}|v^{n-1/2}|\Delta t)^{-1}\left(v^{n-\frac{1}{2}} + \Delta t m^{-1}\left(F^n - s(u^n)\right)\right).$$

The initial conditions are derived at the end of Section 5.4:

$$u^0 = I,$$

$$v^{\frac{1}{2}} = V - \frac{1}{2}\Delta t\omega^2 I\,.$$

7 Exercises and Problems

Problem 1: Use linear/quadratic functions for verific

Consider the ODE problem

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

Discretize this equation according to $[D_t D_t u + \omega^2 u = f]^n$.

a) Derive the equation for the first time step (u^1).

b) For verification purposes, we use the method of manufactured s((MMS) with the choice of $u_{\mathrm{e}}(x,t) = ct + d$. Find restrictions on c and the initial conditions. Compute the corresponding source term f b Show that $[D_t D_t t]^n = 0$ and use the fact that the $D_t D_t$ operator i: $[D_t D_t(ct + d)]^n = c[D_t D_t t]^n + [D_t D_t d]^n = 0$, to show that u_{e} is also a solution of the discrete equations.

c) Use `sympy` to do the symbolic calculations above. Here is a sketc program `vib_undamped_verify_mms.py`:

```
import sympy as sm
V, t, I, w, dt = sm.symbols('V t I w dt')  # global symbols
f = None  # global variable for the source term in the ODE

def ode_source_term(u):
    """Return the terms in the ODE that the source term
    must balance, here u'' + w**2*u.
    u is symbolic Python function of t."""
    return sm.diff(u(t), t, t) + w**2*u(t)

def residual_discrete_eq(u):
    """Return the residual of the discrete eq. with u inserted."""
    R = ...
    return sm.simplify(R)

def residual_discrete_eq_step1(u):
    """Return the residual of the discrete eq. at the first
    step with u inserted."""
    R = ...
```

```
    return sm.simplify(R)

def DtDt(u, dt):
    """Return 2nd-order finite difference for u_tt.
    u is a symbolic Python function of t.
    """
    return ...

def main(u):
    """
    Given some chosen solution u (as a function of t, implemented
    as a Python function), use the method of manufactured solutions
    to compute the source term f, and check if u also solves
    the discrete equations.
    """
    print '=== Testing exact solution: %s ===' % u
    print "Initial conditions u(0)=%s, u'(0)=%s:" % \
          (u(t).subs(t, 0), sm.diff(u(t), t).subs(t, 0))

    # Method of manufactured solution requires fitting f
    global f  # source term in the ODE
    f = sm.simplify(ode_lhs(u))

    # Residual in discrete equations (should be 0)
    print 'residual step1:', residual_discrete_eq_step1(u)
    print 'residual:', residual_discrete_eq(u)

def linear():
    main(lambda t: V*t + I)

if __name__ == '__main__':
    linear()
```

ill in the various functions such that the calls in the `main` function works.

) The purpose now is to choose a quadratic function $u_e = bt^2 + ct + d$ as exact olution. Extend the `sympy` code above with a function `quadratic` for fitting `f` nd checking if the discrete equations are fulfilled. (The function is very similar o `linear`.)

) Will a polynomial of degree three fulfill the discrete equations?

) Implement a `solver` function for computing the numerical solution of this roblem.

) Write a nose test for checking that the quadratic solution is computed o correctly (too machine precision, but the round-off errors accumulate and ıcrease with T) by the `solver` function.

Filenames: `vib_undamped_verify_mms.pdf`, `vib_undamped_verify_mms.py`.

Exercise 2: Show linear growth of the phase with time

Consider an exact solution $I\cos(\omega t)$ and an approximation $I\cos(\tilde{\omega}t)$. Define ıe phase error as time lag between the peak I in the exact solution and the orresponding peak in the approximation after m periods of oscillations. Show ıat this phase error is linear in m. Filename: `vib_phase_error_growth.pdf`.

Exercise 3: Improve the accuracy by adjusting th quency

According to (18), the numerical frequency deviates from the exact frequ a (dominating) amount $\omega^3\Delta t^2/24 > 0$. Replace the `w` parameter in the al in the `solver` function in `vib_undamped.py` by `w*(1 - (1./24)*w**2` and test how this adjustment in the numerical algorithm improves the a (use $\Delta t = 0.1$ and simulate for 80 periods, with and without adjustmen

Filename: `vib_adjust_w.py`.

Exercise 4: See if adaptive methods improve the pha ror

Adaptive methods for solving ODEs aim at adjusting Δt such that the within a user-prescribed tolerance. Implement the equation $u'' + u =$ Odespy[15] software. Use the example from Section **??** in [**?**]. Run the with a very low tolerance (say 10^{-14}) and for a long time, check the nu time points in the solver's mesh (`len(solver.t_all)`), and compare th error with that produced by the simple finite difference method from Sec with the same number of (equally spaced) mesh points. The question is it pays off to use an adaptive solver or if equally many points with a method gives about the same accuracy. Filename: `vib_undamped_adapt`

Exercise 5: Use a Taylor polynomial to compute u^1

As an alternative to the derivation of (8) for computing u^1, one can use a polynomial with three terms for u^1:

$$u(t_1) \approx u(0) + u'(0)\Delta t + \frac{1}{2}u''(0)\Delta t^2$$

With $u'' = -\omega^2 u$ and $u'(0) = 0$, show that this method also leads Generalize the condition on $u'(0)$ to be $u'(0) = V$ and compute u^1 in t with both methods. Filename: `vib_first_step.pdf`.

Exercise 6: Find the minimal resolution of an oscil function

Sketch the function on a given mesh which has the highest possible fr That is, this oscillatory "cos-like" function has its maxima and mi every two grid points. Find an expression for the frequency of this f and use the result to find the largest relevant value of $\omega\Delta t$ when frequency of an oscillating function and Δt is the mesh spacing. Fi `vib_largest_wdt.pdf`.

[15] `https://github.com/hplgit/odespy`

Exercise 7: Visualize the accuracy of finite differences for cosine function

We introduce the error fraction

$$E = \frac{[D_t D_t u]^n}{u''(t_n)}$$

to measure the error in the finite difference approximation $D_t D_t u$ to u''. Compute for the specific choice of a cosine/sine function of the form $u = \exp(i\omega t)$ and show that

$$E = \left(\frac{2}{\omega\Delta t}\right)^2 \sin^2(\frac{\omega\Delta t}{2}).$$

Plot E as a function of $p = \omega\Delta t$. The relevant values of p are $[0, \pi]$ (see Exercise 6 for why $p > \pi$ does not make sense). The deviation of the curve from unity visualizes the error in the approximation. Also expand E as a Taylor polynomial in p up to fourth degree (use, e.g., `sympy`). Filename: `vib_plot_fd_exp_error.py`.

Exercise 8: Verify convergence rates of the error in energy

We consider the ODE problem $u'' + \omega^2 u = 0$, $u(0) = I$, $u'(0) = V$, for $t \in (0, T]$. The total energy of the solution $E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2 u^2$ should stay constant. The error in energy can be computed as explained in Section 5.2.

Make a nose test in a file `test_error_conv.py`, where code from `vib_undamped.py` is imported, but the `convergence_rates` and `test_convergence_rates` functions are copied and modified to also incorporate computations of the error in energy and the convergence rate of this error. The expected rate is 2. Filename: `test_error_conv.py`.

Exercise 9: Use linear/quadratic functions for verification

This exercise is a generalization of Problem 1 to the extended model problem (3) where the damping term is either linear or quadratic. Solve the various subproblems and see how the results and problem settings change with the generalized ODE in case of linear or quadratic damping. By modifying the code from Problem 1, `sympy` will do most of the work required to analyze the generalized problem. Filename: `vib_verify_mms.py`.

Exercise 10: Use an exact discrete solution for verification

Write a nose test function in a separate file that employs the exact discrete solution (19) to verify the implementation of the `solver` function in the file `vib_undamped.py`. Just import `solver` and make functions for the exact discrete solution and the nose test. Filename: `vib_verify_discrete_omega.py`.

Exercise 11: Use analytical solution for convergence rate tests

The purpose of this exercise is to perform convergence tests of the problem (3) when $s(u) = \omega^2 u$ and $F(t) = A\sin\phi t$. Find the complete analytical solution to the problem in this case (most textbooks on mechanics list the

various elements you need to write down the exact solution). Mod `convergence_rate` function from the `vib_undamped.py` program to experiments with the extended model. Verify that the error is of or Filename: `vib_conv_rate.py`.

Exercise 12: Investigate the amplitude errors of many

Use the program `vib_undamped_odespy.py` from Section 5.1 and the tude estimation from the `amplitudes` function in the `vib_undamped` (see Section 3.4) to investigate how well famous methods for 1st-orde can preserve the amplitude of u in undamped oscillations. Test, for e the 3rd- and 4th-order Runge-Kutta methods (`RK3`, `RK4`), the Crank-N method (`CrankNicolson`), the 2nd- and 3rd-order Adams-Bashforth r (`AdamsBashforth2`, `AdamsBashforth3`), and a 2nd-order Backwards (`Backward2Step`). The relevant governing equations are listed in Sec Filename: `vib_amplitude_errors.py`.

Exercise 13: Minimize memory usage of a vibration s

The program `vib.py`[16] store the complete solution $u^0, u^1, \ldots, u^{N_t}$ in r which is convenient for later plotting. Make a memory minimizing ve this program where only the last three u^{n+1}, u^n, and u^{n-1} values are in memory. Write each computed (t_{n+1}, u^{n+1}) pair to file. Visualize t in the file (a cool solution is to read one line at a time and plot the using the line-by-line plotter in the `visualize_front_ascii` functio technique makes it trivial to visualize very long time simulations). Fi `vib_memsave.py`.

Exercise 14: Implement the solver via classes

Reimplement the `vib.py` program using a class `Problem` to hold all the parameters of the problem, a class `Solver` to hold the numerical parame compute the solution, and a class `Visualizer` to display the solution.

Hint. Use the ideas and examples from Section ?? and ?? in [?] specifically, make a superclass `Problem` for holding the scalar physical par of a problem and let subclasses implement the $s(u)$ and $F(t)$ functions as r Try to call up as much existing functionality in `vib.py` as possible.

Filename: `vib_class.py`.

Exercise 15: Show equivalence between schemes

Show that the schemes from Sections 1.2, 5.3, and 5.4 are all equivalent. F `vib_scheme_equivalence.pdf`.

[16] `http://tinyurl.com/jvzzcfn/vib/vib.py`

Exercise 16: Interpret $[D_tD_tu]^n$ as a forward-backward difference

Show that the difference $[D_tD_tu]^n$ is equal to $[D_t^+D_t^-u]^n$ and $D_t^-D_t^+u]^n$. That is, instead of applying a centered difference twice one can alternatively apply a mixture forward and backward differences. Filename: `vib_DtDt_fw_bw.pdf`.

Exercise 17: Use the forward-backward scheme with quadratic damping

We consider the generalized model with quadratic damping, expressed as a system of two first-order equations as in Section 6.8:

$$
\begin{aligned}
u' &= v,\\
v' &= \frac{1}{m}\left(F(t) - \beta|v|v - s(u)\right) .
\end{aligned}
$$

However, contrary to what is done in Section 6.8, we want to apply the idea of the forward-backward discretization in Section 5.3. Express the idea in operator notation and write out the scheme. Unfortunately, the backward difference for the v equation creates a nonlinearity $|v^{n+1}|v^n$. To linearize this nonlinearity, use the known value v^n inside the absolute value factor, i.e., $|v^{n+1}|v^n \approx |v^n|v^{n+1}$. Show that the resulting scheme is equivalent to the one in Section 6.8 for some time level $n \geq 1$.

What we learn from this exercise is that the first-order differences and the linearization trick play together in "the right way" such that the scheme is as good as when we (in Section 6.8) carefully apply centered differences and a geometric mean on a staggered mesh to achieve second-order accuracy. There is a difference in the handling of the initial conditions, though, as explained at the end of Section 5.3. Filename: `vib_gen_bwdamping.pdf`.

Exercise 18: Use a backward difference for the damping term

As an alternative to discretizing the damping terms $\beta u'$ and $\beta|u'|u'$ by centered differences, we may apply backward differences:

$$
\begin{aligned}
[u']^n &\approx [D_t^-u]^n,\\
[|u'|u']^n &\approx [|D_t^-u|D_t^-u]^n = |[D_t^-u]^n|[D_t^-u]^n .
\end{aligned}
$$

The advantage of the backward difference is that the damping term is evaluated using known values u^n and u^{n-1} only. Extend the `vib.py`[17] code with a scheme based on using backward differences in the damping terms. Add statements to compare the original approach with centered difference and the new idea

[17] `http://tinyurl.com/jvzzcfn/vib/vib.py`

launched in this exercise. Perform numerical experiments to investigate how much accuracy that is lost by using the backward differences.

Filename: `vib_gen_bwdamping.pdf`.

References

ndex