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

Aug 27, 2013

Note: **VERY PRELIMINARY VERSION!** (Still lots of typos!)

Contents

List of exercises

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

1 Finite difference discretization

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

1.1 A basic model for vibrations

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

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

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

$$u(t) = I\cos(\omega t)\,. \tag{2}$$

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

In vibrating mechanical systems modeled by (1), $u(t)$ very often represents a position or a displacement of a particular point in the system. The derivative $u'(t)$ then has the interpretation of the point's velocity, and $u''(t)$ is the associated acceleration. The model (1) is not only applicable to vibrating mechanical 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 follow the four steps from Section 1.2 in [**?**].

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

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

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

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

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

Inserting (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 difference. Here we choose a centered difference:

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

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

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

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

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

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

which reduces to

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

Exercise 6 asks you to perform an alternative derivation and also to generalize the initial condition to $u'(0) = V \neq 0$.

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]
```

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

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

Operator notation. We may write the scheme using the compact difference notation (see Section 1.7 in [?]). The difference (4) has the operator notation $[D_tD_tu]^n$ such that we can write:

$$[D_tD_tu + \omega^2 u = 0]^n \,. \tag{9}$$

Note that $[D_tD_tu]^n$ means applying a central difference with step $\Delta t/2$ twice:

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

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 notation be expressed as

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

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

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

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

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

2 Implementation

2.1 Making a solver function

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

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

def solver(I, w, dt, T):
    """
    Solve u'' + w**2*u = 0 for t in (0,T], u(0)=I and u'(0)=0,
```

```
    by a central finite difference method with time step dt.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = zeros(Nt+1)
    t = linspace(0, Nt*dt, Nt+1)

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

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

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

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

A corresponding main program calling these functions for a simulation of a given number 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 be handy. Here is a code segment using the `ArgumentParser` tool in the `argparse` module 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
```

2.2 Verification

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

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

Checking convergence rates. Empirical computation of convergence rates, as explained in Section 2.8 in [**?**], yields a good method for verification. The function 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 $i = 1, \ldots, m-1$.

All the implementational details appear below.

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

```
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
```

The returned `r` list has its values equal to 2.00, which is in excellent agreement with what is expected from the second-order finite difference approximation $[D_tD_tu]^n$ and other theoretical measures of the error in the numerical method. 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 $\Delta t = 0.1, 0.05$ 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 incorrect positions compared with the peaks of the exact cosine solution. This effect can be understood as if also the numerical solution is on the form $I\cos\tilde{\omega}t$, but where $\tilde{\omega}$ is not exactly equal to ω. Later, we shall mathematically quantify this numerical frequency $\tilde{\omega}$.

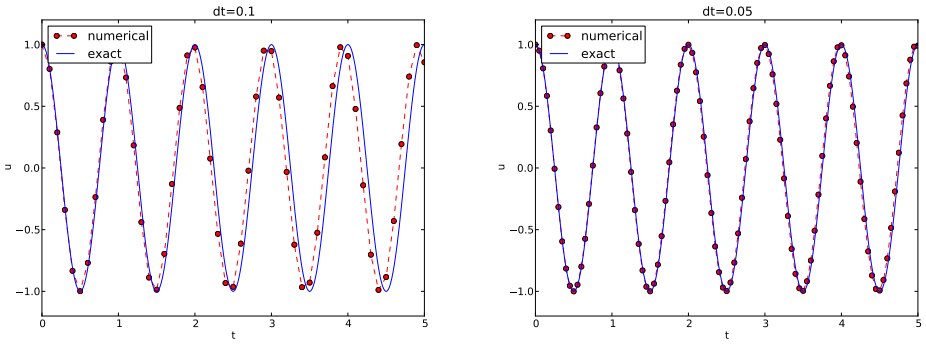


Figure 1: Effect of halving the time step.

3.1 Using a moving plot window

In vibration problems it is often of interest to investigate the system's behavior over long time intervals. Errors in the phase may then show up as crucial. Let us investigate long time series by introducing a moving plot window that can move along with the p most recently computed periods of the solution. The SciTools[3] package contains a convenient tool for this: `MovingPlotWindow`. Typing `pydoc scitools.MovingPlotWindow` shows a demo and description of usage. The function below illustrates the usage and is invoked in the `vib_undamped.py` code 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:
```

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

```
                filename = 'tmp_vib%04d.png' % n
                st.savefig(filename)
                print 'making plot file', filename, 'at t=%g' % t[n]
        plot_manager.update(n)
```

Running

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

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

3.2 Making a movie file

The `visualize_front` function stores all the plots in files whose names are numbered: `tmp_vib0000.png`, `tmp_vib0001.png`, `tmp_vib0002.png`, and so on. From 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 the above command if desired. Other formats can be generated by changing the video 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`, `gxine`, or `totem`, e.g.,

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

A web page can also be used to play the movie. Today's standard is to use the 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 order, it is important to number the files with zero-padded numbers (0000, 0001, 0002, etc.). The printf format `%04d` specifies an integer in a field of width 4, padded with zeros from the left. A simple Unix wildcard file specification like `tmp_vib*.png` will then list the frames in the right order. If the numbers in the filenames were not zero-padded, the frame `tmp_vib11.png` would appear before `tmp_vib2.png` in the movie.

3.3 Using a line-by-line ascii plotter

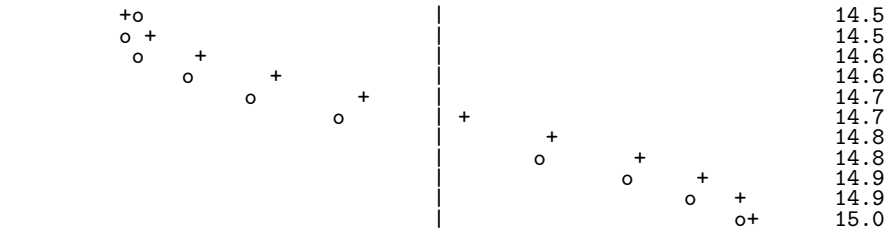
Plotting functions vertically, line by line, in the terminal window using ascii characters only is a simple, fast, and convenient visualization technique for long time series (the time arrow points downward). The tool `scitools.avplotter.Plotter` makes it easy to create such plots:

```
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))
```

The call `p.plot` returns a line of text, with the t axis marked and a symbol `+` for the first function (`u`) and `o` for the second function (the exact solution). Here we append this text a time counter reflecting how many periods the current time point 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
```



3.4 Empirical analysis of the solution

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

The local maxima are the points where

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

and the local minima are recognized by

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

In 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 points, where t_i is the time value and e_i the corresponding u value. The local period 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 since we probably want to compute with it, e.g., find the corresponding frequencies `2*pi/p`.

Having the minima and the maxima, the local amplitude can be calculated as the difference between two neighboring minimum and maximum points:

```
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 conveniently done with just a counter on the horizontal axis, since `a[i]` and `p[i]` correspond to the i-th amplitude estimate and the i-th period estimate, respectively. There is no unique time point associated with either of these estimate since values at two different time points were used in the computations.

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

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 section, we shall now quantify this error through mathematical analysis. The key tool in the analysis will be to establish an exact solution of the discrete equations. The difference equation (7) has constant coefficients and is homogeneous. The solution is then of the form $u^n = A^n$, where A is some number to be determined (recall that n in u^n is a superscript labeling the time level, while n in A^n is an exponent). With oscillating functions as solutions, the algebra will be considerably simplified if we write

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

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

We have

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

The physically relevant numerical solution can be taken as the real part of this complex expression.

[4] `https://github.com/hplgit/INF5620/blob/gh-pages/src/vib/vib_empirical_analysis.py`

The calculations goes as

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

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

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

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

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

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

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

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

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

$$
\tilde{\omega} = \pm\frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right)\,. \tag{17}
$$

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

```
>>> 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
```

This means that

$$
\tilde{\omega} = \omega\left(1 + \frac{1}{24}\omega^2\Delta t^2\right) + \mathcal{O}(\Delta t^3)\,. \tag{18}
$$

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

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

Figure 2 plots the discrete frequency (17) and its approximation (18) for $\omega = 1$ (based on the program `vib_plot_freq.py`[6]). Although $\tilde{\omega}$ is a function of Δt in (18), it is misleading to think of Δt as the important discretization parameter. It is the product $\omega\Delta t$ is the key discretization parameter. This quantity reflects the *number of time steps per period* of the oscillations. To see this, we set $P = N_P \Delta t$, where P is the length of a period, and N_P is the number of time steps during a period. Since P and ω are related by $P = 2\pi/\omega$, we get 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 necessary for reasonable accuracy, but this depends on the length of the simulation (T) as the total phase error due to the frequency error grows linearly with time (see Exercise 3).

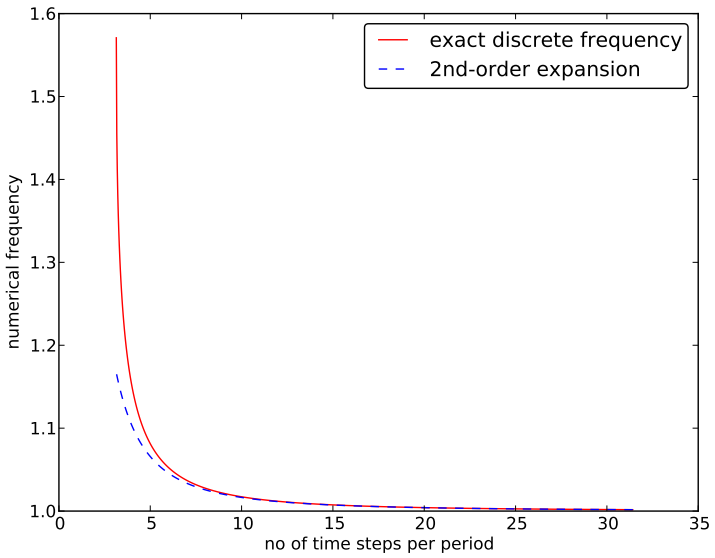


Figure 2: Exact discrete frequency and its second-order series expansion.

[6] `https://github.com/hplgit/INF5620/blob/gh-pages/src/vib/vib_plot_freq.py`

4.2 Exact discrete solution

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

$$u^n = I\cos\left(\tilde{\omega} n\Delta t\right), \quad \tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right). \tag{19}$$

Such an exact discrete solution is ideal for verification purposes (and you are encouraged to make a test based on (19) in Exercise 10).

4.3 Stability

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

We realize that a constant amplitude demands $\tilde{\omega}$ to be a real number. A complex $\tilde{\omega}$ is indeed possible if the argument x of $\sin^{-1}(x)$ has magnitude larger than unity: $|x| > 1$ (type `asin(x)` in wolframalpha.com[7] to see basic properties of $\sin^{-1}(x)$). A complex $\tilde{\omega}$ can be written $\tilde{\omega} = \tilde{\omega}_r + i\tilde{\omega}_i$. Since $\sin^{-1}(x)$ has a *negative* imaginary part for $x > 1$, $\tilde{\omega}_i < 0$, it means that $\exp\left(i\omega\tilde{t}\right) = \exp\left(-\tilde{\omega}_i t\right)\exp\left(i\tilde{\omega}_r t\right)$ will lead to exponential growth in time because $\exp\left(-\tilde{\omega}_i t\right)$ with $\tilde{\omega}_i < 0$ has a positive exponent.

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

$$\frac{\omega\Delta t}{2} \leq 1 \quad \Rightarrow \quad \Delta t \leq \frac{2}{\omega}. \tag{20}$$

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

Summary.

From the analysis we can draw three important conclusions:

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

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

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

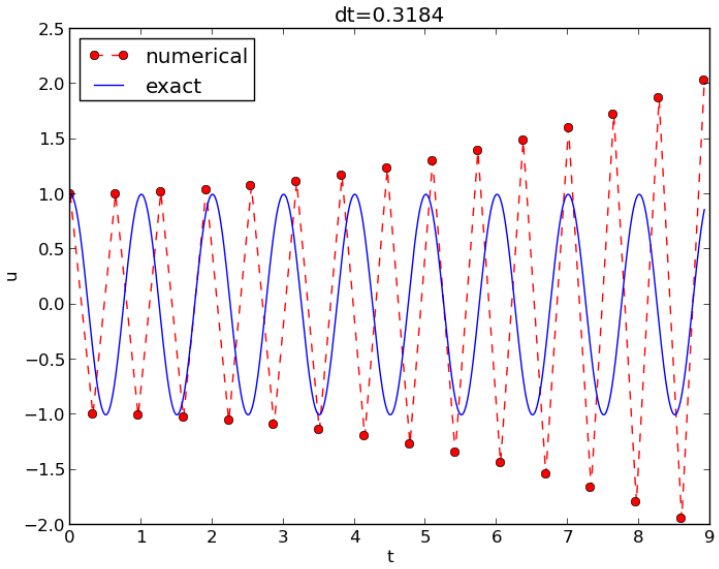


Figure 3: Growing, unstable solution because of a time step slightly beyond the stability limit.

3. The numerical solution exhibits a relative phase error $\tilde{\omega}/\omega \approx 1 + \frac{1}{24}p^2$. This error leads to wrongly displaced peaks of the numerical solution, and the error in peak location grows linearly with time (see Exercise 3).

5 Alternative schemes based on 1st-order equations

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

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

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

$$u' = v, \tag{21}$$

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

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

5.1 Standard methods for 1st-order ODE systems

The Forward Euler scheme. A Forward Euler approximation to our 2×2 system of ODEs (21)-(22) becomes

$$[D_t^+ u = v]^n, [D_t^+ v = -\omega^2 u]^n, \tag{23}$$

or written out,

$$u^{n+1} = u^n + \Delta t v^n, \tag{24}$$
$$v^{n+1} = v^n - \Delta t \omega^2 u^n \, . \tag{25}$$

Let us briefly compare this Forward Euler method with the centered difference scheme for the second-order differential equation. We have from (24) and (25) applied at levels n and $n-1$ that

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

Since from (24)

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

it follows that

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

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

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

$$[D_t^- u = v]^{n+1}, \tag{26}$$
$$[D_t^- v = -\omega u]^{n+1} \, . \tag{27}$$

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

$$u^{n+1} - \Delta t v^{n+1} = u^n, \tag{28}$$
$$v^{n+1} + \Delta t \omega^2 u^{n+1} = v^n \, . \tag{29}$$

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

$$[D_t u = \overline{v}^t]^{n+\frac{1}{2}}, \tag{30}$$

$$[D_t v = -\omega \overline{u}^t]^{n+\frac{1}{2}} . \tag{31}$$

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

$$u^{n+1} - \frac{1}{2}\Delta t v^{n+1} = u^n + \frac{1}{2}\Delta t v^n, \tag{32}$$

$$v^{n+1} + \frac{1}{2}\Delta t \omega^2 u^{n+1} = v^n - \frac{1}{2}\Delta t \omega^2 u^n . \tag{33}$$

Comparison of schemes. We can easily compare methods like the ones above (and many more!) with the aid of the Odespy[8] package. Below is a possible program for doing that.

```
import odespy
import numpy as np

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

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

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

There is quite some more code dealing with plots also, and we refer to the source file `vib_odespy.py`[9] for details. Observe that keyword arguments in `f(u,t,w=1)` can be supplied through a solver parameter `f_kwargs` (dictionary).

Specification of the Forward Euler, Backward Euler, and Crank-Nicolson schemes 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'),
    ]
```

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

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

The `vib_odespy.py`[10] program makes two plots of the computed solutions with the various methods in the `solvers` list: one plot with $u(t)$ versus t, and one *phase plane plot* where v is plotted against u. That is, the phase plane plot is the curve $(u(t), v(t))$ parameterized by t. Analytically, $u = I\cos(\omega t)$ and $v = u' = -\omega I\sin(\omega t)$. The exact curve $(u(t), v(t))$ is therefore an ellipse, which often looks like a circle in a plot if the axes are automatically scaled. The important feature, however, is that exact curve $(u(t), v(t))$ is closed and repeats itself for every period. Not all numerical schemes are capable to do that, meaning that the amplitude instead shrinks or grows with time.

The Forward Euler scheme in Figure 4 has a pronounced spiral curve, pointing to the fact that the amplitude steadily grows, which is also evident in Figure 5. The Backward Euler scheme has a similar feature, except that the spriral goes inward and the amplitude is significantly damped. The changing amplitude and the sprial form decreases with decreasing time step. The Crank-Nicolson scheme looks much more accurate. In fact, these plots tell that the Forward and Backward Euler schemes are not suitable for solving our ODEs with oscillating solutions.

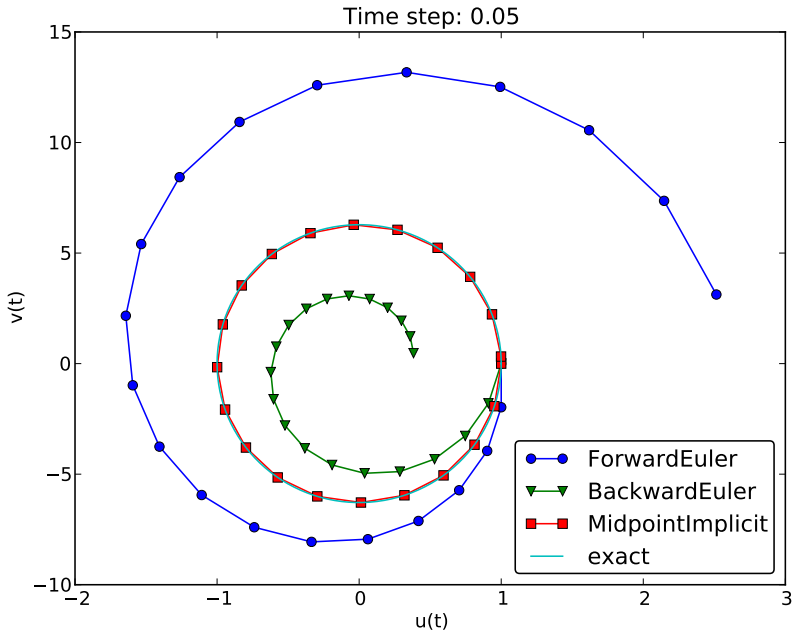


Figure 4: Comparison of classical schemes in the phase plane.

We may run two popular standard methods for first-order ODEs, the 2nd- and 4th-order Runge-Kutta methods, to see how they perform. Figures 6 and 7

[10] `https://github.com/hplgit/INF5620/blob/gh-pages/src/vib/vib_odespy.py`

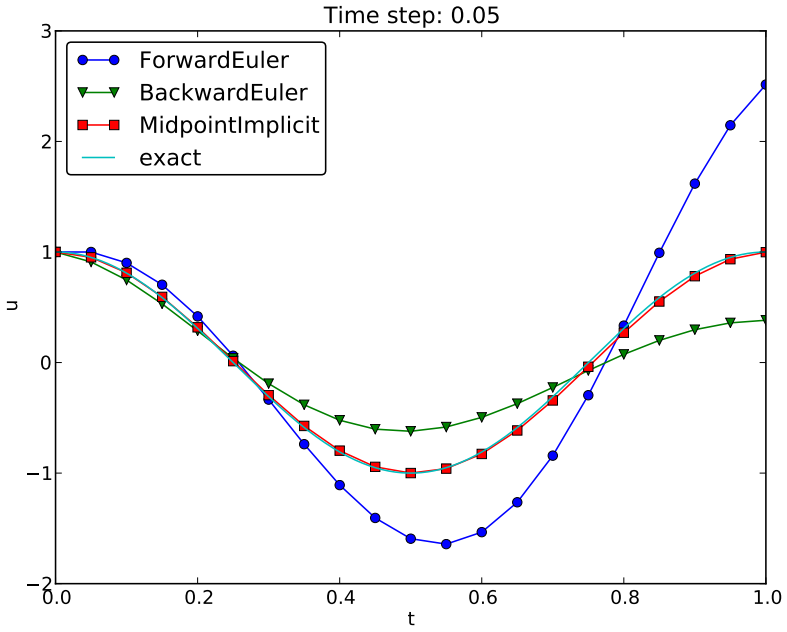


Figure 5: Comparison of classical schemes.

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

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

The corresponding results for the Crank-Nicolson scheme are shown in Figures 8 and 9. It is clear that the Crank-Nicolson scheme outperforms the 2nd-order Runge-Kutta method. Both schemes have the same order of accuracy $\mathcal{O}(\Delta t^2)$, but their differences in the accuracy that matters in a real physical application is very clearly pronounced in this example. Exercise 12 invites you to investigate how

5.2 The Euler-Cromer method

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

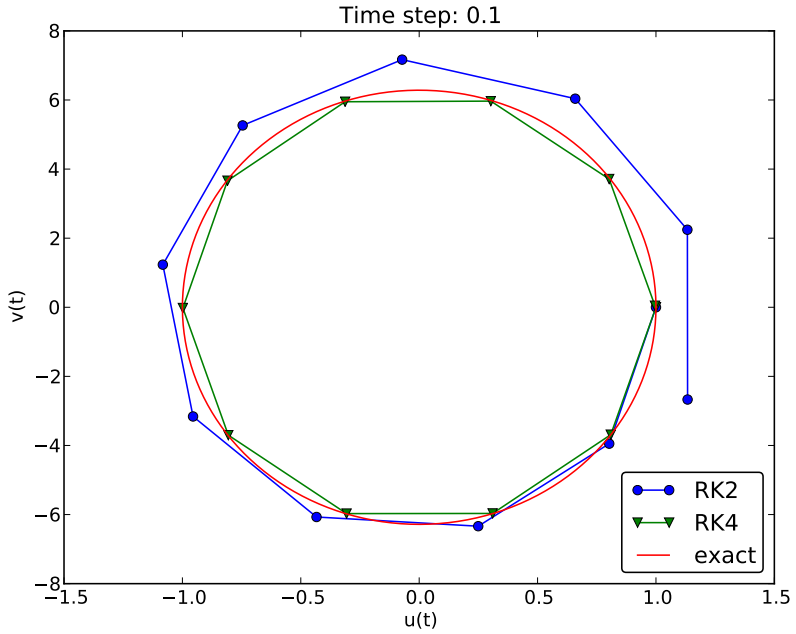


Figure 6: Comparison of Runge-Kutta schemes in the phase plane.

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

$$[D_t^+ u = v]^n, \tag{34}$$
$$[D_t^- v = -\omega u]^{n+1}\,. \tag{35}$$

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

$$u^{n+1} = u^n + \Delta t v^n, \tag{36}$$
$$v^{n+1} = v^n - \Delta t \omega^2 u^{n+1}\,. \tag{37}$$

We realize that u^{n+1} can be computed from (36) and then v^{n+1} from (37) using the recently computed value u^{n+1} on the right-hand side.

The scheme (36)-(37) goes under several names: Forward-backward scheme, Semi-implicit Euler method[11], symplectic Euler, semi-explicit Euler, Newton-Störmer-Verlet, and Euler-Cromer. We shall stick to the latter name. Since both time discretizations are based on first-order difference approximation, one may

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

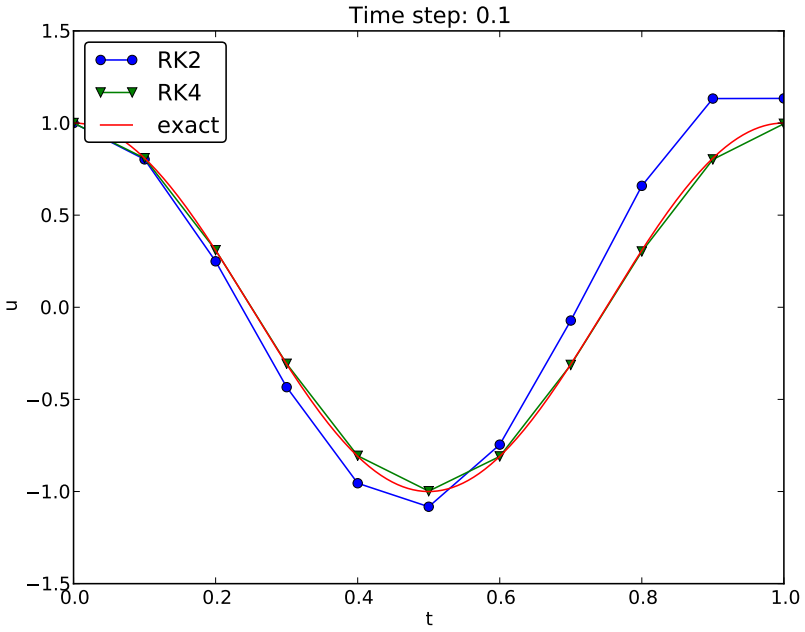


Figure 7: Comparison of Runge-Kutta schemes.

think that the scheme is only of first-order, but this is not true: the use of a forward and then a backward difference make errors cancel so that the overall error in the scheme is $\mathcal{O}(\Delta t^2)$. This is explaned below.

Equivalence with the scheme for the second-order ODE. We may eliminate the v^n variable from (36)-(37). From (37) we have $v^n = v^{n-1} - \Delta t \omega^2 u^n$, which can be inserted in (36) to yield

$$u^{n+1} = u^n + \Delta t v^{n-1} - \Delta t^2 \omega^2 u^n. \tag{38}$$

The v^{n-1} quantity can be expressed by u^n and u^{n-1} using (36):

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

and when this is inserted in (38) we get

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2 \omega^2 u^n, \tag{39}$$

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

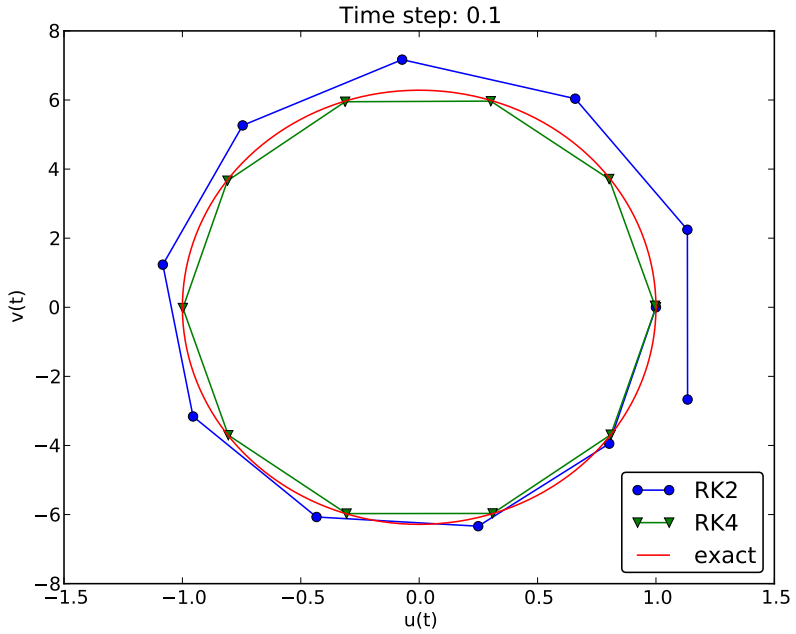


Figure 8: Long-time behavior of the Crank-Nicolson scheme in the phase plane.

The initial condition $u' = 0$ means $u' = v = 0$. Then $v^0 = 0$, and (36) implies $u^1 = u^0$, while (37) says $v^1 = -\omega^2 u^0$. This approximation, $u^1 = u^0$, corresponds to a first-order Forward Euler discretization of the initial condition $u'(0) = 0$: $[D_t^+ u = 0]^0$. Therefore, the Euler-Cromer scheme will start out differently and not exactly reproduce the solution of (7).

5.3 A method utilizing a staggered mesh

The Forward and Backward Euler schemes used in the Euler-Cromer method are both unsymmetric, but their combination yields a symmetric method. This symmetry is much more evident if we introduce a *staggered mesh* in time where u is sought at integer time points t_n and v is sought at $t_{n+1/2}$ between two u points. The unknowns are then $u^1, v^{3/2}, u^2, v^{5/2}$, and so on.

We can now easily use centered difference approximations, expressed in operator notation as

$$[D_t u = v]^{n+\frac{1}{2}}, \tag{40}$$

$$[D_t v = -\omega u]^{n+1}\,. \tag{41}$$

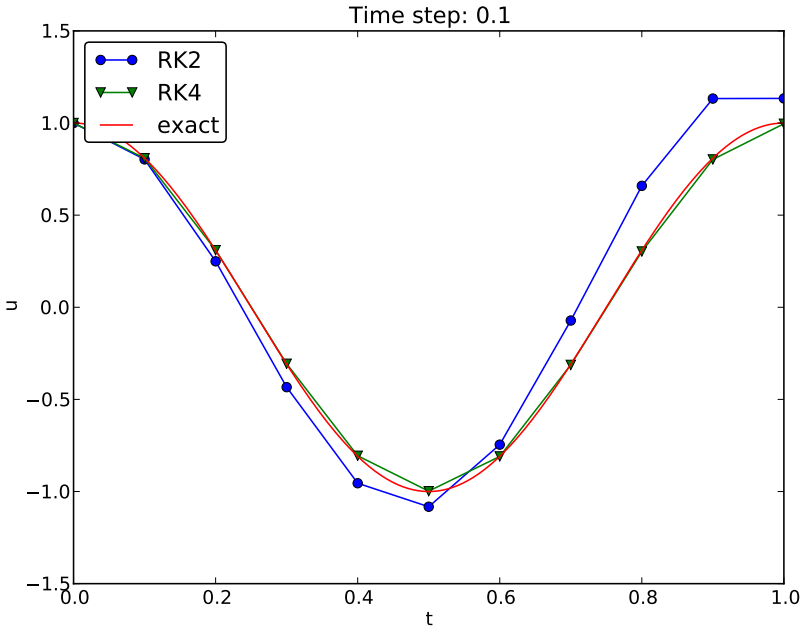


Figure 9: Long-time behavior of the Crank-Nicolson scheme.

Writing out the formulas gives

$$u^{n+1} = u^n + \Delta t v^{n+\frac{1}{2}}, \tag{42}$$

$$v^{n+\frac{3}{2}} = v^{n+\frac{1}{2}} - \Delta t \omega^2 u^{n+1}\,. \tag{43}$$

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

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

$$v^{n+\frac{1}{2}} = v^{n-\frac{1}{2}} - \Delta t \omega^2 u^n\,. \tag{45}$$

Such a rewrite is only mathematical cosmetics. The important thing is that this centered scheme has exactly the same computational structure as the forward-backward scheme. We shall use the names *forward-backward Euler-Cromer* and *staggered Euler-Cromer* to distinguish the two schemes. In the latter case we can eliminate the v values and get back the centered scheme based on the second-order differential equation, so all these three schemes are equivalent. However, they differ somewhat in the treatment of the initial conditions.

Suppose we have $u(0) = I$ and $u'(0) = v(0) = 0$ as mathematical initial 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 (45) 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 scheme based on the second-order differential equation (and hence corresponds to a centered difference approximation of the initial condition for $u'(0)$). The conclusion is that a staggered mesh is fully equivalent with that scheme, while the forward-backward version gives a slight deviation in the computation of u^1.

We realize that u^{n+1} can be computed from (36) and then v^{n+1} from (37) using the recently computed value u^{n+1} on the right-hand side.

6 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 $F(t)$:

$$mu'' + f(u') + s(u) = F(t), \quad u(0) = I,\ u'(0) = V,\ t \in (0, T]\,. \tag{46}$$

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 $s(u) \sim \sin(u)$.

6.1 A centered scheme for linear damping

Sampling (46) 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

$$[mD_tD_tu + f(D_{2t}u) + s(u) = F]^n, \tag{47}$$

which written out means

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + f(\frac{u^{n+1} - u^{n-1}}{2\Delta t}) + s(u^n) = F^n, \tag{48}$$

where F^n as usual means $F(t)$ evaluated at $t = t_n$. Solving (48) 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

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + b\frac{u^{n+1} - u^{n-1}}{2\Delta t} + s(u^n) = F^n, \tag{49}$$

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

$$u^{n+1} = \left(2mu^n + (\frac{b}{2}\Delta t - m)u^{n-1} + \Delta t^2(F^n - s(u^n))\right)(m + \frac{b}{2}\Delta t)^{-1}\,. \tag{50}$$

For the first time step we need to discretize $u'(0) = V$ as $[D_{2t}u = V]^0$ and combine with (50) for $n = 0$. The discretized initial condition leads to

$$u^{-1} = u^1 - 2\Delta t V, \tag{51}$$

which inserted in (50) for $n = 0$ gives an equation that can be solved for u^1:

$$u^1 = u^0 + \Delta t\, V + \frac{\Delta t^2}{2m}(-bV - s(u^0) + F^0)\,. \tag{52}$$

6.2 A centered scheme for quadratic damping

When $f(u') = bu'|u'|$, we get a quadratic equation for u^{n+1} in (48). This equation can straightforwardly be solved, but we can also avoid the nonlinearity by performing an approximation that is within other numerical errors that we have already committed by replacing derivatives with finite differences.

The idea is to reconsider (46) and only replace u'' by D_tD_tu, giving

$$[mD_tD_tu + bu'|u'| + s(u) = F]^n, \tag{53}$$

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

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

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

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

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

$$u'(t_{n+1/2}) \approx [D_tu]^{n+1/2}, \quad u'(t_{n-1/2}) \approx [D_tu]^{n-1/2}\,. \tag{54}$$

We then get

$$[u'|u'|]^n \approx [D_tu]^{n+1/2}[|D_tu|]^{n-1/2} = \frac{u^{n+1} - u^n}{\Delta t}\frac{|u^n - u^{n-1}|}{\Delta t}\,. \tag{55}$$

The counterpart to (48) is then

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

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

$$u^{n+1}=\left(m+b|u^n-u^{n-1}|\right)^{-1}\times \left(2mu^n-mu^{n-1}+bu^n|u^n-u^{n-1}|+\Delta t^2(F^n-s(u^n))\right). \tag{57}$$

For $n=0$ we run into some trouble: inserting (51) in (57) results in a complicated nonlinear equation for u^1. By thinking differently about the problem we can get away with the nonlinearity (again). We have for $n=0$ that $b[u'|u'|]^0 = bV|V|$. Using this value in (53) gives

$$[mD_tD_tu+bV|V|+s(u)=F]^0. \tag{58}$$

Writing this equation out and using (51) gives

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

7 Implementation

7.1 Algorithm and solver function

The algorithm is very similar to the undamped case. The difference is basically a question of different formulas for u^1 and u^{n+1}. This is actually quite remarkable. The equation (46) is normally impossible to solve by pen and paper, but possible for some special choices of F, s, and f. On the contrary, the nonlinear generalized model (46) versus the simple undamped model does not make much sense when we solve the problem numerically!

The computational algorithm is a slight variation of the one in Section 1.2:

1. $u^0=I$
2. compute u^1 from (52) if linear damping or (59) if quadratic damping
3. for $n=1,2,\ldots,N_t-1$:
   (a) compute u^{n+1} from (vib:ode2:step4) if linear damping or (57) if quadratic damping

Modifying the `solver` function for the undamped case is fairly easy, the big 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 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
```

7.2 Verification

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

7.3 Visualization

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

The `vib.py`[12] code contains several functions for analyzing the time series signal and for visualizing the solutions.

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

7.4 User interface

The `main` function has substantial changes from the `vib_undamped.py` code since we need to specify the new data c, $s(u)$, and $F(t)$. In addition, we must set T and the plot window width (instead of the number of periods we want to simulate as in `vib_undamped.py`). To figure out whether we can use one plot for the whole time series or if we should follow the most recent part of u, we can use the `plot_empricial_freq_and_amplitude` function's estimate of the number of local maxima. This number is now returned from the function and used 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 the model problem (46). As a demo of `vib.py`, we consider the case $I = 1$, $V = 0$, $m = 1$, $c = 0.03$, $s(u) = \sin(u)$, $F(t) = 3\cos(4t)$, $\Delta t = 0.05$, and $T = 140$. The relevant 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[13] on the screen. Figure 10 shows a part of the time series.

[13] `http://tinyurl.com/k3sdbuv/notes/mov-vib/vib_generalized_dt0.05/index.html`

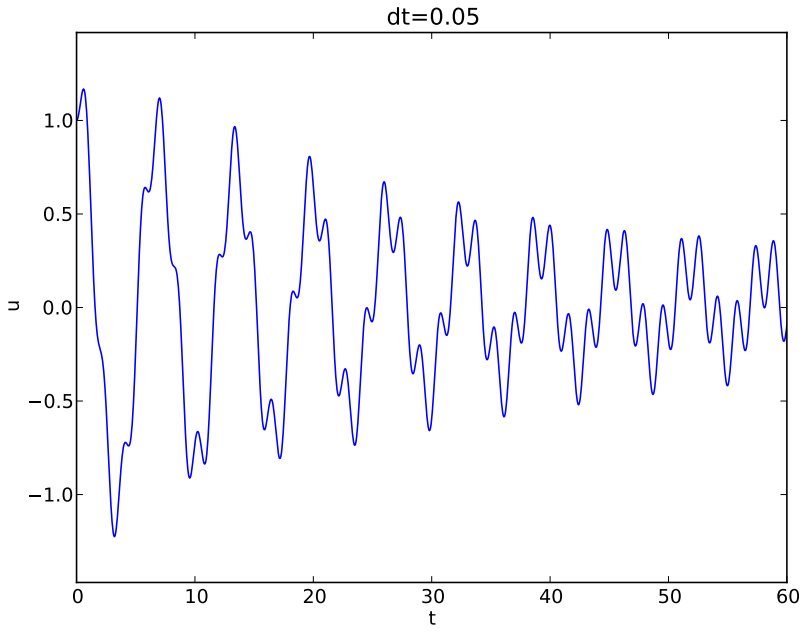


Figure 10: Damped oscillator excited by a sinusoidal function.

8 Exercises

Exercise 1: Use a linear function for verification

Add a source term to the ODE problem (1), $u' + \omega^2 u = f(t)$, and a more general initial condition $u'(0) = V$, where V is a given constant. Discretize this equation according to $[D_t D_t u + \omega^2 u = f]_i^n$ and derive a new equation for the first time step (u^1). Let $u_{\mathrm{e}}(x,t) = ct + I$ and show that this is a solution of the ODE if $f(x,t) = \omega^2(ct + I)$ and $V = c$. Show that $[D_t D_t t]_i^n = 0$ and therefore that u_{e} *also* solves the discrete equations. Implement the modified problem and make a nose test with the linear u_{e} to verify the solution. Filename: `vib_undamped_verify_linear.py`.

Exercise 2: Use a quadratic function for verification

This is a variation of Exercise 1 where we test a quadratic function rather than a linear. Let $u_{\mathrm{e}}(x,t) = at^2 + ct + I$, find the corresponding f term, and show that u_{e} fulfills all discrete equations. You will need to show that $[D_t D_t t^2]^n = 2$. Filename: `vib_undamped_verify_quadratic.py`.

Exercise 3: 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 the phase error as time lag between the peak I in the exact solution and the corresponding peak in the approximation after m periods of oscillations. Show that this phase error is linear in m. Filename: `vib_phase_error_growth.pdf`.

Exercise 4: Improve the accuracy by adjusting the frequency

According to (18), the numerical frequency deviates from the exact frequency by a (dominating) amount $\omega^3\Delta t^2/24 > 0$. Replace the `w` parameter in the algorithm in the `solver` function (in `vib_undamped.py`) by `w*(1 - (1./24)*w**2*dt**2` and test how this adjustment in the numerical algorithm improves the accuracy. Filename: `vib_adjust_w.py`.

Exercise 5: See if adaptive methods improve the phase error

Adaptive methods for solving ODEs aim at adjusting Δt such that the error is within a user-prescribed tolerance. Implement the equation $u'' + u = 0$ in the Odespy[14] software. Use the example from Section 9.11 in [**?**]. Run the scheme with a very low tolerance (say 10^{-14}) and for a long time, check the number of time points in the solver's mesh (`len(solver.t_all)`), and compare the phase error with that produced by the simple finite difference method from Section 1.2 with the same number of (equally spaced) mesh points. The question is whether it pays off to use an adaptive solver or if equally many points with a simple method gives about the same accuracy. Filename: `vib_undamped_adaptive.py`.

Exercise 6: Use a Taylor polynomial to compute u^1

As an alternative to the derivation of (8) for computing u^1, one can use a Taylor 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 to (8). Generalize the condition on $u'(0)$ to be $u'(0) = V$ and compute u^1 in this case with both methods. Filename: `vib_first_step.pdf`.

Exercise 7: Find the minimal resolution of an oscillatory function

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

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

Exercise 8: Visualize the accuracy of finite differences for a 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 E for the specific choice of a cosine/sine function of the form $u = \exp(i\omega t)$ and show that

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

Plot E as a function of $p = \omega\Delta t$. The relevant values of p are $[0, \pi]$ (see Exercise 7 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 9: Use a linear function for verification

This is a continuation of Exercise 1 where we consider the extended model problem (46). Prescribe $u_{\mathrm{e}} = Vt + I$, which fulfills the initial conditions, and show that this u_{e} solves (46) if $F(t) = I + (c + t)V$. Import the `solver` function from `vib.py` and add a nose test using this linear solution. Filename: `vib_verify_linear.py`.

Exercise 10: Use an exact discrete solution for verification

Write a nose test function that employs the exact discrete solution (19) to verify the implementation of the `solver` function in the file `vib_undamped.py`. 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 (46) 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). Modify the `convergence_rate` function from the `vib_undamped.py` program to perform experiments with the extended model. Verify that the error is of order Δt^2. Filename: `vib_conv_rate.py`.

Exercise 12: Investigate the amplitude errors of many solvers

Use the program `vib_odespy.py` from Section 5.1 and the amplitude estimation from the `amplitudes` function in the `vib_undamped.py` file (see Section 3.4) to investigate how well famous methods for 1st-order ODEs can preserve the amplitude of u in undamped oscillations. Test, for example, the 3rd- and 4th-order Runge-Kutta methods (`RK3`, `RK4`), the Crank-Nicolson method (`CrankNicolson`), the 2nd- and 3rd-order Adams-Bashforth methods

(`AdamsBashforth2`, `AdamsBashforth3`), and a 2nd-order Backwards scheme (`Backward2Step`). The relevant governing equations are listed in Section 21. Filename: `vib_amplitude_errors.py`.

Exercise 13: Minimize memory usage of a vibration solver

The program `vib.py`[15] store the complete solution $u^0, u^1, \ldots, u^{N_t}$ in memory, which is convenient for later plotting. Make a memory minimizing version of this program where only the last three u^{n+1}, u^n, and u^{n-1} values are stored in memory. Write each computed (t_{n+1}, u^{n+1}) pair to file. Visualize the data in the file (a cool solution is to read one line at a time and plot the u value using the line-by-line plotter in the `visualize_front_ascii` function - this technique makes it trivial to visualize very long time simulations). Filename: `vib_memsave.py`.

Exercise 14: Implement the solver via classes

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

Hint. Use the ideas and examples from Section 3.6 and 3.7 in [**?**]. More specifically, make a superclass `Problem` for holding the scalar physical parameters of a problem and let subclasses implement the $s(u)$ and $F(t)$ functions as methods. Try to call up as much existing functionality in `vib.py` as possible. Filename: `vib_class.py`.

[15] `https://github.com/hplgit/INF5620/blob/gh-pages/src/vib/vib.py`

Index