Finite Difference Computing for Oscillatory Phenomena

Hans Petter Langtangen[1,2]

Svein Linge[3,1]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo
[3]Department of Process, Energy and Environmental Technology, University College of Southeast Norway

Jan 21, 2017

Contents

List of Exercises and Problems

Vibration problems lead to differential equations with solutions that oscillate 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 or very smooth. Both the frequency and amplitude of the oscillations need to be accurately handled by the numerical schemes. The forthcoming text presents a range of different methods, from classical ones (Runge-Kutta and midpoint/Crank-Nicolson methods), to more modern and popular symplectic (geometric) integration schemes (Leapfrog, Euler-Cromer, and Störmer-Verlet methods), but with a clear emphasis on the latter. Vibration problems occur throughout mechanics and physics, but the methods discussed in this text are also fundamental for constructing successful algorithms for partial differential equations of wave nature in multiple spatial dimensions.

1 Finite difference discretization

Many of the numerical challenges faced when computing oscillatory solutions to ODEs and PDEs can be captured by the very simple ODE $u'' + u = 0$. This ODE is thus chosen as our starting point for method development, implementation, and analysis.

1.1 A basic model for vibrations

The simplest model of a vibrating mechanical system has the following form:

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

Here, ω and I are given constants. Section 12.1 derives (1) from physical principles and explains what the constants mean.

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 is 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 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 explained in Section 1.1.2 in [2].

Step 1: Discretizing the domain. The domain is discretized by introducing a uniformly partitioned time mesh. The points in the mesh are $t_n = n\Delta t$, $n = 0, 1, \ldots, N_t$, 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. (Note that $n = 0$ is the known initial condition, so u^n is identical to the mathematical u at this point.) The mesh function u^n 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 where the solution must be found:

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

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, whose accuracy is similar to the centered difference we used for u'':

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

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 to be solved for:

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

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, Verlet integration, or the Leapfrog method (one should note that Leapfrog is used for many quite different methods for quite different differential equations!).

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 comes to our rescue: (6) implies $u^{-1} = u^1$ and this relation can be combined with (7) for $n = 0$ 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 5 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) become

1. $u^0 = I$
2. compute u^1 from (8)
3. for $n = 1, 2, \ldots, N_t - 1$: 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 on using `w` for ω in computer code.

In the code, we use `w` as the symbol for ω. The reason is that the authors prefer `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 a compact difference notation (see also Section 1.1.8 in [2]). The difference (4) has the operator notation $[D_t D_t u]^n$ such that we can write:

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

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

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

which is written out as

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

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

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

2 Implementation

2.1 Making a solver function

The algorithm from the previous section is readily translated to a complete Python function for computing and 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:

```
import numpy as np
import matplotlib.pyplot as plt

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 = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)

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

We have imported `numpy` and `matplotlib` under the names `np` and `plt`, respectively, as this is very common in the Python scientific computing community and a good programming habit (since we explicitly see where the different functions

come from). An alternative is to do `from numpy import *` and a similar "import all" for Matplotlib to avoid the `np` and `plt` prefixes and make the code as close as possible to MATLAB. (See Section 5.1.4 in [2] for a discussion of the two types of import in Python.)

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

```
def u_exact(t, I, w):
    return I*np.cos(w*t)

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

A corresponding main program calling these functions to simulate 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 via 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
```

Such parsing of the command line is explained in more detail in Section 5.2.3 in [2].

A typical execution goes like

Terminal

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

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

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

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

$$v^n = [D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} \,.$$

Typical (scalar) code is

```
v = np.zeros_like(u)  # or v = np.zeros(len(u))
# Use central difference for internal points
for i in range(1, len(u)-1):
    v[i] = (u[i+1] - u[i-1])/(2*dt)
# Use initial condition for u'(0) when i=0
v[0] = 0
# Use backward difference at the final mesh point
v[-1] = (u[-1] - u[-2])/dt
```

Since the loop is slow for large N_t, we can get rid of the loop by vectorizing the central difference. The above code segment goes as follows in its vectorized version (see Problem 1.2 in [2] for explanation of details):

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

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. The `test_three_steps` function in the file `vib_undamped.py` shows the details of how we use the hand calculations to test the code:

```
def test_three_steps():
    from math import pi
    I = 1;  w = 2*pi;  dt = 0.1;  T = 1
    u_by_hand = np.array([1.000000000000000,
                          0.802607911978213,
                          0.288358920740053])
    u, t = solver(I, w, dt, T)
    diff = np.abs(u_by_hand - u[:3]).max()
    tol = 1E-14
    assert diff < tol
```

This function is a proper *test function*, compliant with the pytest and nose testing framework for Python code, because

- the function name begins with `test_`
- the function takes no arguments
- the test is formulated as a boolean condition and executed by `assert`

See Section 5.3.2 in [2] for more details on how to construct test functions and utilize nose or pytest for automatic execution of tests. Our recommendation is to use pytest. With this choice, you can run all test functions in `vib_undamped.py` by

Terminal

```
Terminal> py.test -s -v vib_undamped.py
============================= test session starts ======...
platform linux2 -- Python 2.7.9 -- ...
collected 2 items

vib_undamped.py::test_three_steps PASSED
vib_undamped.py::test_convergence_rates PASSED

========================== 2 passed in 0.19 seconds ===...
```

Testing very simple polynomial 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 $f = 2 + (\omega t)^2$. By simple insertion we can show that the mesh function $u^n = t_n^2$ is also a solution of the discrete equations. Problem 1 asks you to carry out all details to show that linear and quadratic solutions are solutions of the discrete equations. Such results are very useful for debugging and verification. You are strongly encouraged to do this problem now!

Checking convergence rates. Empirical computation of convergence rates yields a good method for verification. The method and its computational details are explained in detail in Section 3.1.6 in [2]. Readers not familiar with the concept should look up this reference before proceeding.

In the present problem, computing convergence rates means that we must

- perform m simulations, halving the time steps as: $\Delta t_i = 2^{-i}\Delta t_0$, $i = 1, \ldots, m-1$, and Δt_i is the time step used in simulation i;
- compute the L^2 norm of the error, $E_i = \sqrt{\Delta t_i \sum_{n=0}^{N_t-1}(u^n - u_{\mathrm{e}}(t_n))^2}$ in each case;
- estimate the convergence rates r_i based on two consecutive experiments $(\Delta t_{i-1}, E_{i-1})$ and $(\Delta t_i, E_i)$, assuming $E_i = C(\Delta t_i)^r$ and $E_{i-1} = C(\Delta t_{i-1})^r$, where C is a constant. From these equations it follows that $r = \ln(E_{i-1}/E_i)/\ln(\Delta t_{i-1}/\Delta t_i)$. Since this r will vary with i, we equip it with an index and call it r_{i-1}, where i runs from 1 to $m-1$.

The computed rates $r_0, r_1, \ldots, r_{m-2}$ hopefully converge to the number 2 in the present problem, because theory (from Section 4) shows that the error of the numerical method we use behaves like Δt^2. The convergence of the sequence $r_0, r_1, \ldots, r_{m-2}$ demands that the time steps Δt_i are sufficiently small for the error model $E_i = C(\Delta t_i)^r$ to be valid.

All the implementational details of computing the sequence $r_0, r_1, \ldots, r_{m-2}$ appear below.

```
def convergence_rates(m, solver_function, num_periods=8):
    """
    Return m-1 empirical estimates of the convergence rate
    based on m simulations, where the time step is halved
    for each simulation.
    solver_function(I, w, dt, T) solves each problem, where T
    is based on simulation for num_periods periods.
    """
    from math import pi
    w = 0.35; I = 0.3       # just chosen values
    P = 2*pi/w              # period
    dt = P/30               # 30 time step per period 2*pi/w
    T = P*num_periods

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

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

The error analysis in Section 4 is quite detailed and suggests that $r = 2$. It is also a intuitively reasonable result, since we used a second-order accurate finite difference approximation $[D_t D_t u]^n$ to the ODE and a second-order accurate finite difference formula for the initial condition for u'.

In the present problem, when Δt_0 corresponds to 30 time steps per period, the returned `r` list has all its values equal to 2.00 (if rounded to two decimals). This amazingly accurate result means that all Δt_i values are well into the asymptotic regime where the error model $E_i = C(\Delta t_i)^r$ is valid.

We can now construct a proper test function that computes convergence rates and checks that the final (and usually the best) estimate is sufficiently close to 2. Here, a rough tolerance of 0.1 is enough. Later, we will argue for an improvement by adjusting omega and include also that case in our test function here. The unit test goes like

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

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

Visualizing convergence rates with slope markers. Tony S. Yu has written a script `plotslopes.py` that is very useful to indicate the slope of a graph, especially a graph like $\ln E = r \ln \Delta t + \ln C$ arising from the model $E = C\Delta t^r$. A copy of the script resides in the `src/vib` directory. Let us use it to compare the original method for $u'' + \omega^2 u = 0$ with the same method applied to the equation with a modified ω. We make log-log plots of the error versus Δt. For each curve we attach a slope marker using the `slope_marker((x,y), r)` function from `plotslopes.py`, where `(x,y)` is the position of the marker and `r` and the slope $((r, 1))$, here (2,1) and (4,1).

```
def plot_convergence_rates():
    r2, E2, dt2 = convergence_rates(
        m=5, solver_function=solver, num_periods=8)
    plt.loglog(dt2, E2)
    r4, E4, dt4 = convergence_rates(
```

```
        m=5, solver_function=solver_adjust_w, num_periods=8)
    plt.loglog(dt4, E4)
    plt.legend(['original scheme', r'adjusted $\omega$'],
               loc='upper left')
    plt.title('Convergence of finite difference methods')
    from plotslopes import slope_marker
    slope_marker((dt2[1], E2[1]), (2,1))
    slope_marker((dt4[1], E4[1]), (4,1))
```

Figure 1 displays the two curves with the markers. The match of the curve slope and the marker slope is excellent.

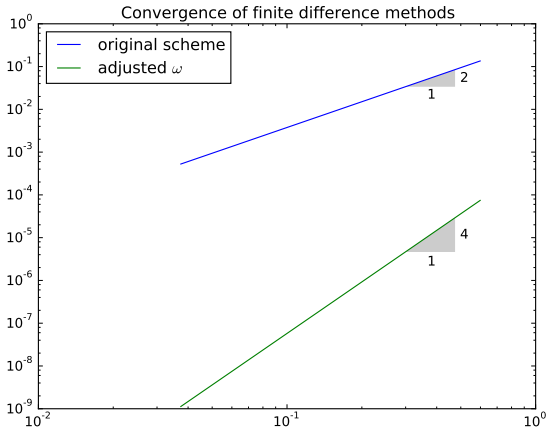


Figure 1: Empirical convergence rate curves with special slope marker.

2.3 Scaled model

It is advantageous to use dimensionless variables in simulations, because fewer parameters need to be set. The present problem is made dimensionless by introducing dimensionless variables $\bar{t} = t/t_c$ and $\bar{u} = u/u_c$, where t_c and u_c are characteristic scales for t and u, respectively. We refer to Section 2.2.1 in [3] for all details about this scaling.

The scaled ODE problem reads

$$\frac{u_c}{t_c^2}\frac{d^2\bar{u}}{d\bar{t}^2} + u_c\bar{u} = 0, \quad u_c\bar{u}(0) = I, \ \frac{u_c}{t_c}\frac{d\bar{u}}{d\bar{t}}(0) = 0\,.$$

A common choice is to take t_c as one period of the oscillations, $t_c = 2\pi/w$, and $u_c = I$. This gives the dimensionless model

$$\frac{d^2\bar{u}}{d\bar{t}^2} + 4\pi^2\bar{u} = 0, \quad \bar{u}(0) = 1, \ \bar{u}'(0) = 0\,. \tag{13}$$

Observe that there are no physical parameters in (13)! We can therefore perform a single numerical simulation $\bar{u}(\bar{t})$ and afterwards recover any $u(t;\omega,I)$ by

$$u(t;\omega,I) = u_c\bar{u}(t/t_c) = I\bar{u}(\omega t/(2\pi))\,.$$

We can easily check this assertion: the solution of the scaled problem is $\bar{u}(\bar{t}) = \cos(2\pi\bar{t})$. The formula for u in terms of $\bar{u}$ gives $u = I\cos(\omega t)$, which is nothing but the solution of the original problem with dimensions.

The scaled model can by run by calling `solver(I=1, w=2*pi, dt, T)`. Each period is now 1 and `T` simply counts the number of periods. Choosing `dt` as `1./M` gives `M` time steps per period.

3 Visualization of long time simulations

Figure 2 shows a comparison of the exact and numerical solution for the scaled model (13) with $\Delta t = 0.1, 0.05$. From the plot we make the following observations:

- The numerical solution seems to have correct amplitude.
- There is an angular frequency error which is reduced by decreasing the time step.
- The total angular frequency error grows with time.

By angular frequency error we mean that the numerical angular frequency differs from the exact ω. This is evident by looking at the peaks of the numerical solution: these have incorrect positions compared with the peaks of the exact cosine solution. The effect can be mathematically expressed by writing the numerical solution as $I\cos\tilde{\omega}t$, where $\tilde{\omega}$ is not exactly equal to ω. Later, we shall mathematically quantify this numerical angular frequency $\tilde{\omega}$.

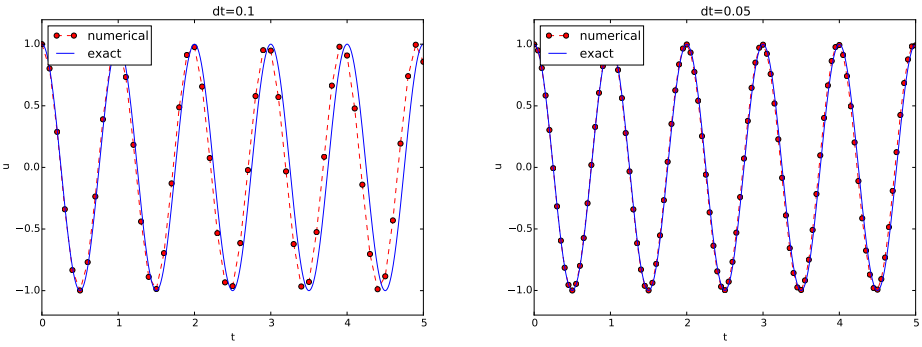


Figure 2: Effect of halving the time step.

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 angular frequency accumulate and become more visible as time grows. We can 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 package contains a convenient tool for this: `MovingPlotWindow`. Typing `pydoc scitools.MovingPlotWindow` shows a demo and a description of its use. The function below utilizes the moving plot window and is in fact called by the `main` function in the `vib_undamped` module if the number of periods in the simulation exceeds 10.

```
def visualize_front(u, t, I, w, savefig=False, skip_frames=1):
    """
    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.
    Plots are saved to files if savefig is True.
    Only each skip_frames-th plot is saved (e.g., if
    skip_frame=10, only each 10th plot is saved to file;
    this is convenient if plot files corresponding to
    different time steps are to be compared).
    """
    import scitools.std as st
    from scitools.MovingPlotWindow import MovingPlotWindow
    from math import pi

    # Remove all old plot files tmp_*.png
    import glob, os
    for filename in glob.glob('tmp_*.png'):
        os.remove(filename)

    P = 2*pi/w  # one period
    umin = 1.2*u.min();  umax = -umin
    dt = t[1] - t[0]
    plot_manager = MovingPlotWindow(
        window_width=8*P,
        dt=dt,
        yaxis=[umin, umax],
        mode='continuous drawing')
    frame_counter = 0
    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 and n % skip_frames == 0:
                filename = 'tmp_%04d.png' % frame_counter
                st.savefig(filename)
                print 'making plot file', filename, 'at t=%g' % t[n]
                frame_counter += 1
        plot_manager.update(n)
```

We run the scaled problem (the default values for the command-line arguments `-I` and `-w` correspond to the scaled problem) for 40 periods with 20 time steps per period:

Terminal

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

The moving plot window is invoked, and we can follow the numerical and exact solutions as time progresses. From this demo we see that the angular frequency error is small in the beginning, and that it becomes more prominent with time. A new run with $\Delta t = 0.1$ (i.e., only 10 time steps per period) clearly shows that the phase errors become significant even earlier in the time series, deteriorating the solution further.

3.2 Making animations

Producing standard video formats. The `visualize_front` function stores all the plots in files whose names are numbered: `tmp_0000.png`, `tmp_0001.png`, `tmp_0002.png`, and so on. From these files we may make a movie. The Flash format is popular,

Terminal

```
Terminal> ffmpeg -r 25 -i tmp_%04d.png -c:v flv movie.flv
```

The `ffmpeg` program can be replaced by the `avconv` program in the above command if desired (but at the time of this writing it seems to be more momentum in the `ffmpeg` project). The `-r` option should come first and describes the number of frames per second in the movie (even if we would like to have slow movies, keep this number as large as 25, otherwise files are skipped from the movie). The `-i` option describes the name of the plot files. Other formats can be generated by changing the video codec and equipping the video file with the right extension:

Format	Codec and filename
Flash	`-c:v flv movie.flv`
MP4	`-c:v libx264 movie.mp4`
WebM	`-c:v libvpx movie.webm`
Ogg	`-c:v libtheora movie.ogg`

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

Terminal

```
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>
```

Modern browsers do not support all of the video formats. MP4 is needed to successfully play the videos on Apple devices that use the Safari browser. WebM is the preferred format for Chrome, Opera, Firefox, and Internet Explorer v9+. Flash was a popular format, but older browsers that required Flash can play MP4. All browsers that work with Ogg can also work with WebM. This means that to have a video work in all browsers, the video should be available in the MP4 and WebM formats. The proper HTML code reads

```
<video autoplay loop controls
       width='640' height='365' preload='none'>
<source src='movie.mp4'   type='video/mp4;
 codecs="avc1.42E01E, mp4a.40.2"'>
<source src='movie.webm'  type='video/webm;
 codecs="vp8, vorbis"'>
</video>
```

The MP4 format should appear first to ensure that Apple devices will load the video correctly.

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_*.png` will then list the frames in the right order. If the numbers in the filenames were not zero-padded, the frame `tmp_11.png` would appear before `tmp_2.png` in the movie.

Playing PNG files in a web browser. The `scitools movie` command can create a movie player for a set of PNG files such that a web browser can be used to watch the movie. This interface has the advantage that the speed of the movie can easily be controlled, a feature that scientists often appreciate. The command for creating an HTML with a player for a set of PNG files `tmp_*.png` goes like

Terminal

```
Terminal> scitools movie output_file=vib.html fps=4 tmp_*.png
```

The `fps` argument controls the speed of the movie ("frames per second").

To watch the movie, load the video file `vib.html` into some browser, e.g.,

Terminal

```
Terminal> google-chrome vib.html  # invoke web page
```

Clicking on `Start movie` to see the result. Moving this movie to some other place requires moving `vib.html` *and all the PNG files* `tmp_*.png`:

Terminal

```
Terminal> mkdir vib_dt0.1
Terminal> mv tmp_*.png vib_dt0.1
Terminal> mv vib.html vib_dt0.1/index.html
```

Making animated GIF files. The `convert` program from the ImageMagick software suite can be used to produce animated GIF files from a set of PNG files:

Terminal

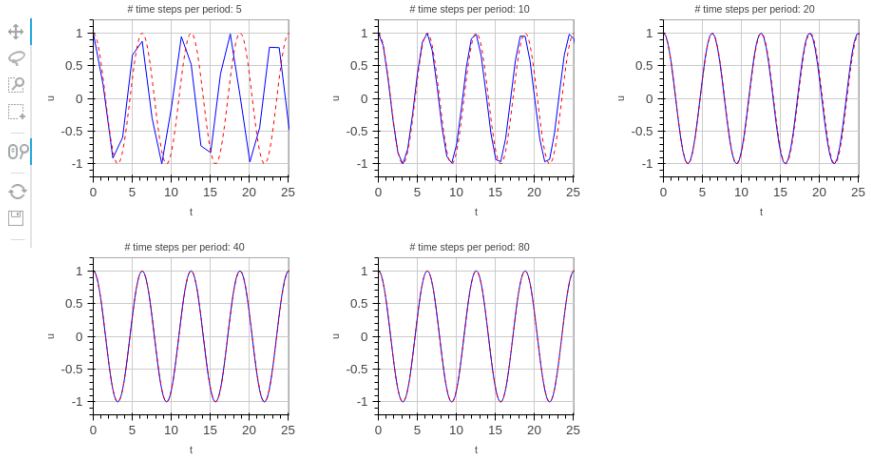
```
Terminal> convert -delay 25 tmp_vib*.png tmp_vib.gif
```

The `-delay` option needs an argument of the delay between each frame, measured in 1/100 s, so 4 frames/s here gives 25/100 s delay. Note, however, that in this particular example with $\Delta t = 0.05$ and 40 periods, making an animated GIF file out of the large number of PNG files is a very heavy process and not considered feasible. Animated GIFs are best suited for animations with not so many frames and where you want to see each frame and play them slowly.

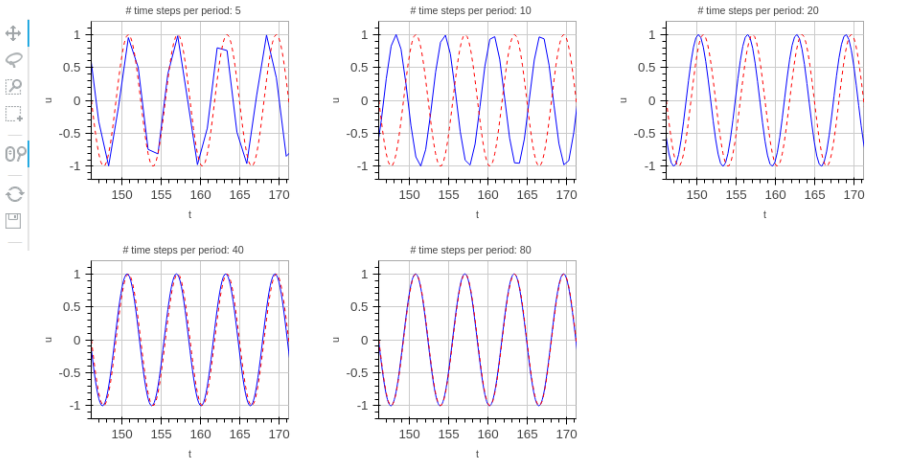
3.3 Using Bokeh to compare graphs

Instead of a moving plot frame, one can use tools that allow panning by the mouse. For example, we can show four periods of several signals in several plots and then scroll with the mouse through the rest of the simulation *simultaneously* in all the plot windows. The Bokeh plotting library offers such tools, but the plots must be displayed in a web browser. The documentation of Bokeh is excellent, so here we just show how the library can be used to compare a set of u curves corresponding to long time simulations. (By the way, the guidance to correct pronunciation of Bokeh in the documentation and on Wikipedia is not directly compatible with a YouTube video...).

Imagine we have performed experiments for a set of Δt values. We want each curve, together with the exact solution, to appear in a plot, and then arrange all plots in a grid-like fashion:



Furthermore, we want the axes to couple such that if we move into the future in one plot, all the other plots follows (note the displaced t axes!):



A function for creating a Bokeh plot, given a list of `u` arrays and corresponding `t` arrays, is implemented below. The code combines data from different simulations, described compactly in a list of strings `legends`.

```
def bokeh_plot(u, t, legends, I, w, t_range, filename):
    """
    Make plots for u vs t using the Bokeh library.
    u and t are lists (several experiments can be compared).
    legens contain legend strings for the various u,t pairs.
    """
    if not isinstance(u, (list,tuple)):
        u = [u]  # wrap in list
    if not isinstance(t, (list,tuple)):
```

```
        t = [t]  # wrap in list
    if not isinstance(legends, (list,tuple)):
        legends = [legends]  # wrap in list

    import bokeh.plotting as plt
    plt.output_file(filename, mode='cdn', title='Comparison')
    # Assume that all t arrays have the same range
    t_fine = np.linspace(0, t[0][-1], 1001)  # fine mesh for u_e
    tools = 'pan,wheel_zoom,box_zoom,reset,'\
            'save,box_select,lasso_select'
    u_range = [-1.2*I, 1.2*I]
    font_size = '8pt'
    p = []  # list of plot objects
    # Make the first figure
    p_ = plt.figure(
        width=300, plot_height=250, title=legends[0],
        x_axis_label='t', y_axis_label='u',
        x_range=t_range, y_range=u_range, tools=tools,
        title_text_font_size=font_size)
    p_.xaxis.axis_label_text_font_size=font_size
    p_.yaxis.axis_label_text_font_size=font_size
    p_.line(t[0], u[0], line_color='blue')
    # Add exact solution
    u_e = u_exact(t_fine, I, w)
    p_.line(t_fine, u_e, line_color='red', line_dash='4 4')
    p.append(p_)
    # Make the rest of the figures and attach their axes to
    # the first figure's axes
    for i in range(1, len(t)):
        p_ = plt.figure(
            width=300, plot_height=250, title=legends[i],
            x_axis_label='t', y_axis_label='u',
            x_range=p[0].x_range, y_range=p[0].y_range, tools=tools,
            title_text_font_size=font_size)
        p_.xaxis.axis_label_text_font_size = font_size
        p_.yaxis.axis_label_text_font_size = font_size
        p_.line(t[i], u[i], line_color='blue')
        p_.line(t_fine, u_e, line_color='red', line_dash='4 4')
        p.append(p_)

    # Arrange all plots in a grid with 3 plots per row
    grid = [[]]
    for i, p_ in enumerate(p):
        grid[-1].append(p_)
        if (i+1) % 3 == 0:
            # New row
            grid.append([])
    plot = plt.gridplot(grid, toolbar_location='left')
    plt.save(plot)
    plt.show(plot)
```

A particular example using the `bokeh_plot` function appears below.

```
def demo_bokeh():
    """Solve a scaled ODE u'' + u = 0."""
    from math import pi
    w = 1.0        # Scaled problem (frequency)
    P = 2*np.pi/w  # Period
    num_steps_per_period = [5, 10, 20, 40, 80]
    T = 40*P       # Simulation time: 40 periods
    u = []         # List of numerical solutions
    t = []         # List of corresponding meshes
    legends = []
    for n in num_steps_per_period:
        dt = P/n
        u_, t_ = solver(I=1, w=w, dt=dt, T=T)
        u.append(u_)
        t.append(t_)
        legends.append('# time steps per period: %d' % n)
    bokeh_plot(u, t, legends, I=1, w=w, t_range=[0, 4*P],
               filename='tmp.html')
```

3.4 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. Note that the time axis then is positive downwards on the screen, so we can let the solution be visualized "forever". 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
    from math import pi
    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 to 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
     +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
```

3.5 Empirical analysis of the solution

For oscillating functions like those in Figure 2 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

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

and the local minima are recognized by

$$u^{n-1} > u^n < u^{n+1}, \quad n = 1, \ldots, N_t - 1\,. \tag{15}$$

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 two returned objects are lists 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`.

Since `a[i]` and `p[i]` correspond to the i-th amplitude estimate and the i-th period estimate, respectively, it is most convenient to visualize the `a` and `p` values with the index `i` on the horizontal axis. (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. Let us do this for the scaled problem and $\Delta t = 0.1, 0.05, 0.01$. A ready-made function

```
plot_empirical_freq_and_amplitude(u, t, I, w)
```

computes the empirical amplitudes and periods, and creates a plot where the amplitudes and angular frequencies are visualized together with the exact amplitude `I` and the exact angular frequency `w`. We can make a little program for creating the plot:

```
from vib_undamped import solver, plot_empirical_freq_and_amplitude
from math import pi
dt_values = [0.1, 0.05, 0.01]
u_cases = []
t_cases = []
for dt in dt_values:
    # Simulate scaled problem for 40 periods
    u, t = solver(I=1, w=2*pi, dt=dt, T=40)
```

```
    u_cases.append(u)
    t_cases.append(t)
plot_empirical_freq_and_amplitude(u_cases, t_cases, I=1, w=2*pi)
```

Figure 3 shows the result: we clearly see that lowering Δt improves the angular frequency significantly, while the amplitude seems to be more accurate. The lines with $\Delta t = 0.01$, corresponding to 100 steps per period, can hardly be distinguished from the exact values. The next section shows how we can get mathematical insight into why amplitudes are good while frequencies are more inaccurate.

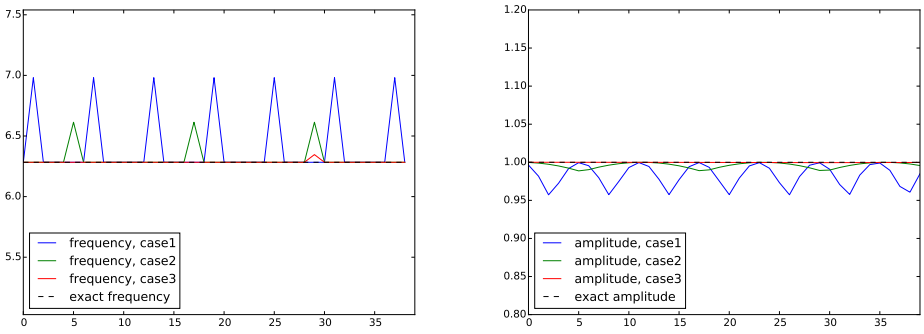


Figure 3: Empirical angular frequency (left) and amplitude (right) for three different time steps.

4 Analysis of the numerical scheme

4.1 Deriving a solution of the numerical scheme

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. Such equations are known to have solutions on the form $u^n = CA^n$, where A is some number to be determined from the difference equation and C is found as the initial condition ($C = I$). 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 seek an A on the form

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

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

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

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

The calculations go as

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

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

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

$$-Ie^{i\tilde{\omega}t_n}\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2}) + \omega^2 Ie^{i\tilde{\omega}t_n} = 0, \tag{16}$$

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

$$\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2}) = \omega^2\,. \tag{17}$$

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

$$\tilde{\omega} = \pm\frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right)\,. \tag{18}$$

4.2 The error in the numerical frequency

The first observation following (18) 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 (18)? 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 (18):

```
>>> 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

$$\tilde{\omega} = \omega\left(1 + \frac{1}{24}\omega^2\Delta t^2\right) + \mathcal{O}(\Delta t^4)\,. \tag{19}$$

The error in the numerical frequency is of second-order in Δt, and the error vanishes as $\Delta t \to 0$. 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 left plot in Figure 2.

Figure 4 plots the discrete frequency (18) and its approximation (19) for $\omega = 1$ (based on the program `vib_plot_freq.py`). Although $\tilde{\omega}$ is a function of Δt in (19), it is misleading to think of Δt as the important discretization parameter. It is the product $\omega\Delta t$ that 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 2).

4.3 Empirical convergence rates and adjusted ω

The expression (19) suggests that adjusting omega to

$$\omega\left(1 - \frac{1}{24}\omega^2\Delta t^2\right),$$

could have effect on the *convergence rate* of the global error in u (cf. Section 2.2). With the `convergence_rates` function in `vib_undamped.py` we can easily check this. A special solver, with adjusted w, is available as the function `solver_adjust_w`. A call to `convergence_rates` with this solver reveals that the rate is 4.0! With the original, physical ω the rate is 2.0 - as expected from using second-order finite difference approximations, as expected from the

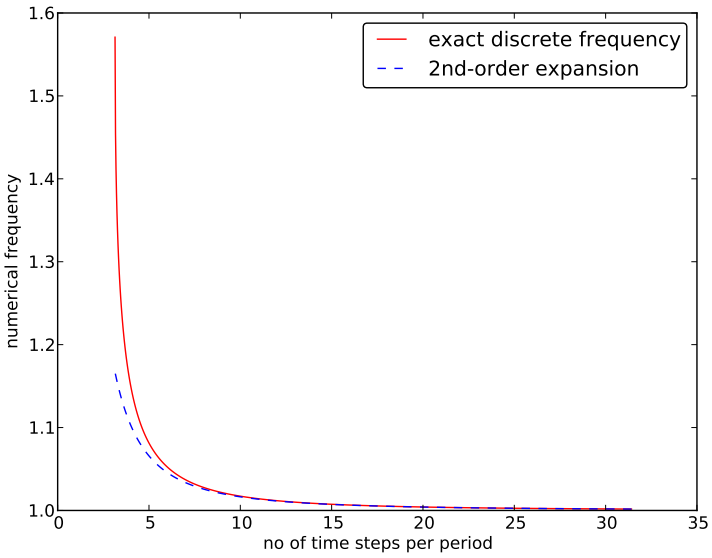


Figure 4: Exact discrete frequency and its second-order series expansion.

forthcoming derivation of the global error, and as expected from truncation error analysis analysis.

Adjusting ω is an ideal trick for this simple problem, but when adding damping and nonlinear terms, we have no simple formula for the impact on ω, and therefore we cannot use the trick.

4.4 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:

$$u^n = I\cos\left(\tilde{\omega} n\Delta t\right), \quad \tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right). \tag{20}$$

We can then compute the error mesh function

$$e^n = u_{\mathrm{e}}(t_n) - u^n = I\cos\left(\omega n\Delta t\right) - I\cos\left(\tilde{\omega} n\Delta t\right). \tag{21}$$

From the formula $\cos 2x - \cos 2y = -2\sin(x-y)\sin(x+y)$ we can rewrite e^n so the expression is easier to interpret:

$$e^n = -2I\sin\left(t\frac{1}{2}\left(\omega - \tilde{\omega}\right)\right)\sin\left(t\frac{1}{2}\left(\omega + \tilde{\omega}\right)\right). \tag{22}$$

The error mesh function is ideal for verification purposes and you are strongly encouraged to make a test based on (20) by doing Exercise 11.

4.5 Convergence

We can use (19) and (21), or (22), to show *convergence* of the numerical scheme, i.e., $e^n \to 0$ as $\Delta t \to 0$, which implies that the numerical solution approaches the exact solution as Δt approaches to zero. We have that

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

by L'Hopital's rule. This result could also been computed WolframAlpha, or we could use the limit functionality in `sympy`:

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

Also (19) can be used to establish that $\tilde{\omega} \to \omega$ when $\Delta t \to 0$. It then follows from the expression(s) for e^n that $e^n \to 0$.

4.6 The global error

To achieve more analytical insight into the nature of the global error, we can Taylor expand the error mesh function (21). Since $\tilde{\omega}$ in (18) contains Δt in the denominator we use the series expansion for $\tilde{\omega}$ inside the cosine function. A relevant `sympy` session is

```
>>> from sympy import *
>>> 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)
>>> w_tilde_series
w + dt**2*w**3/24 + O(dt**4)
```

Series expansions in `sympy` have the inconvenient `O()` term that prevents further calculations with the series. We can use the `removeO()` command to get rid of the `O()` term:

```
>>> w_tilde_series = w_tilde_series.removeO()
>>> w_tilde_series
dt**2*w**3/24 + w
```

Using this `w_tilde_series` expression for $\tilde{w}$ in (21), dropping I (which is a common factor), and performing a series expansion of the error yields

```
>>> 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**6)
```

Since we are mainly interested in the leading-order term in such expansions (the term with lowest power in Δt, which goes most slowly to zero), we use the `.as_leading_term(dt)` construction to pick out this term:

```
>>> error.series(dt, 0, 6).as_leading_term(dt)
dt**2*t*w**3*sin(t*w)/24
```

The last result means that the leading order global (true) error at a point t is proportional to $\omega^3 t\Delta t^2$. Considering only the discrete t_n values for t, t_n is related to Δt through $t_n = n\Delta t$. The factor $\sin(\omega t)$ can at most be 1, so we use this value to bound the leading-order expression to its maximum value

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

This is the dominating term of the error *at a point*.

We are interested in the accumulated global error, which can be taken as the ℓ^2 norm of e^n. The norm is simply computed by summing contributions from all mesh points:

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

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

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

This is our expression for the global (or integrated) error. A primary result from this expression is that the global error is proportional to Δt^2.

4.7 Stability

Looking at (20), it appears that the numerical solution has constant and correct amplitude, but an error in the angular frequency. A constant amplitude is not necessarily the case, however! To see this, note that if only Δt is large enough, the magnitude of the argument to $\sin^{-1}$ in (18) may be larger than 1, i.e., $\omega\Delta t/2 > 1$. In this case, $\sin^{-1}(\omega\Delta t/2)$ has a complex value and therefore $\tilde{\omega}$ becomes complex. Type, for example, `asin(x)` in wolframalpha.com 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 $x > 1$, $\tilde{\omega}_i < 0$, which means that $e^{i\tilde{\omega}t} = e^{-\tilde{\omega}_i t}e^{i\tilde{\omega}_r t}$ will lead to exponential growth in time because $e^{-\tilde{\omega}_i t}$ with $\tilde{\omega}_i < 0$ has a positive exponent.

Stability criterion.

We do not tolerate growth in the amplitude since such growth is not present in the exact solution. Therefore, we must impose a *stability criterion* so that the argument in the inverse sine function leads to real and not complex values of $\tilde{\omega}$. The stability criterion reads

$$\frac{\omega\Delta t}{2} \leq 1 \quad \Rightarrow \quad \Delta t \leq \frac{2}{\omega}\,. \tag{23}$$

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

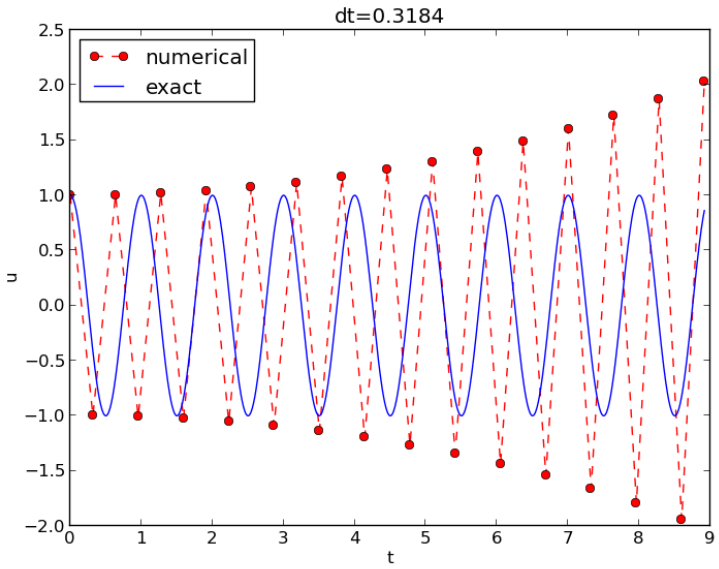


Figure 5: Growing, unstable solution because of a time step slightly beyond the stability limit.

4.8 About the accuracy at the stability limit

An interesting question is whether the stability condition $\Delta t < 2/\omega$ is unfortunate, or more precisely: would it be meaningful to take larger time steps to speed up computations? The answer is a clear no. At the stability limit, we have that $\sin^{-1}\omega\Delta t/2 = \sin^{-1} 1 = \pi/2$, and therefore $\tilde{\omega} = \pi/\Delta t$. (Note that the approximate formula (19) is very inaccurate for this value of Δt as it predicts $\tilde{\omega} = 2.34/pi$, which is a 25 percent reduction.) The corresponding period of the numerical solution is $\tilde{P} = 2\pi/\tilde{\omega} = 2\Delta t$, which means that there is just one time

step Δt between a peak (maximum) and a through (minimum) in the numerical solution. This is the shortest possible wave that can be represented in the mesh! In other words, it is not meaningful to use a larger time step than the stability limit.

Also, the error in angular frequency when $\Delta t = 2/\omega$ is severe: Figure 6 shows a comparison of the numerical and analytical solution with $\omega = 2\pi$ and $\Delta t = 2/\omega = \pi^{-1}$. Already after one period, the numerical solution has a through while the exact solution has a peak (!). The error in frequency when Δt is at the stability limit becomes $\omega - \tilde{\omega} = \omega(1 - \pi/2) \approx -0.57\omega$. The corresponding error in the period is $P - \tilde{P} \approx 0.36P$. The error after m periods is then $0.36mP$. This error has reached half a period when $m = 1/(2 \cdot 0.36) \approx 1.38$, which theoretically confirms the observations in Figure 6 that the numerical solution is a through ahead of a peak already after one and a half period. Consequently, Δt should be chosen much less than the stability limit to achieve meaningful numerical computations.

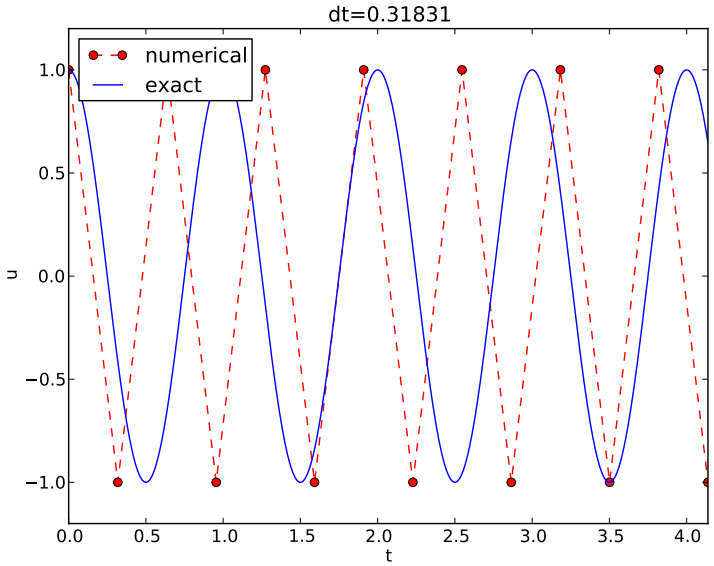


Figure 6: Numerical solution with Δt exactly at the stability limit.

Summary.

From the accuracy and stability 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.

3. The ratio of the numerical angular frequency and the exact one is $\tilde{\omega}/\omega \approx 1 + \frac{1}{24}p^2$. The error $\frac{1}{24}p^2$ leads to wrongly displaced peaks of the numerical solution, and the error in peak location grows linearly with time (see Exercise 2).

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 choose a solution strategy from 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:

$$u' = v, \tag{24}$$
$$v' = -\omega^2 u\,. \tag{25}$$

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

5.1 The Forward Euler scheme

A Forward Euler approximation to our 2×2 system of ODEs (24)-(25) becomes

$$[D_t^+ u = v]^n, \tag{26}$$
$$[D_t^+ v = -\omega^2 u]^n, \tag{27}$$

or written out,

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

Let us briefly compare this Forward Euler method with the centered difference scheme for the second-order differential equation. We have from (28) and (29) 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 (28)

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

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. Rewriting, so that $\Delta t^2 \omega^2 u^{n-1}$ appears alone on the right-hand side, and then dividing by Δt^2, the new left-hand side is an approximation to u'' at t_n, while the right-hand side is sampled at t_{n-1}. All terms should be sampled at the same mesh point, so using $\omega^2 u^{n-1}$ instead of $\omega^2 u^n$ points to a kind of mathematical error in the derivation of the scheme. This error turns out to be rather crucial for the accuracy of the Forward Euler method applied to vibration problems (Section 5.4 has examples).

The reasoning above does not imply that the Forward Euler scheme is not correct, but more that it is almost equivalent to a second-order accurate scheme for the second-order ODE formulation, and that the error committed has to do with a wrong sampling point.

5.2 The Backward Euler scheme

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

$$[D_t^- u = v]^{n+1}, \tag{30}$$
$$[D_t^- v = -\omega u]^{n+1} \,. \tag{31}$$

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

$$u^{n+1} - \Delta t v^{n+1} = u^n, \tag{32}$$
$$v^{n+1} + \Delta t \omega^2 u^{n+1} = v^n \,. \tag{33}$$

We can compare (32)-(33) with the centered scheme (7) for the second-order differential equation. To this end, we eliminate v^{n+1} in (32) using (33) solved

with respect to v^{n+1}. Thereafter, we eliminate v^n using (32) solved with respect to v^{n+1} and also replacing $n+1$ by n and n by $n-1$. The resulting equation involving only u^{n+1}, u^n, and u^{n-1} can be ordered as

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

which has almost the same form as the centered scheme for the second-order differential equation, but the right-hand side is evaluated at u^{n+1} and not u^n. This inconsistent sampling of terms has a dramatic effect on the numerical solution, as we demonstrate in Section 5.4.

5.3 The Crank-Nicolson scheme

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

$$[D_t u = \overline{v}^t]^{n+\frac{1}{2}}, \tag{34}$$

$$[D_t v = -\omega^2 \overline{u}^t]^{n+\frac{1}{2}}\,. \tag{35}$$

Writing the equations out and rearranging terms, shows that this is also a coupled system of two linear equations at each time level:

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

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

We may compare also this scheme to the centered discretization of the second-order ODE. It turns out that the Crank-Nicolson scheme is equivalent to the discretization

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = -\omega^2 \frac{1}{4}(u^{n+1} + 2u^n + u^{n-1}) = -\omega^2 u^n + \mathcal{O}(\Delta t^2)\,. \tag{38}$$

That is, the Crank-Nicolson is equivalent to (7) for the second-order ODE, apart from an extra term of size Δt^2, but this is an error of the same order as in the finite difference approximation on the left-hand side of the equation anyway. The fact that the Crank-Nicolson scheme is so close to (7) makes it a much better method than the Forward or Backward Euler methods for vibration problems, as will be illustrated in Section 5.4.

Deriving (38) is a bit tricky. We start with rewriting the Crank-Nicolson equations as follows

$$u^{n+1} - u^n = \frac{1}{2}\Delta t(v^{n+1} + v^n), \tag{39}$$

$$v^{n+1} = v^n - \frac{1}{2}\Delta t \omega^2 (u^{n+1} + u^n), \tag{40}$$

and add the latter at the previous time level as well:

$$v^n = v^{n-1} - \frac{1}{2}\Delta t\omega^2(u^n + u^{n-1}) \tag{41}$$

We can also rewrite (39) at the previous time level as

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

Inserting (40) for v^{n+1} in (39) and (41) for v^n in (39) yields after some reordering:

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

Now, $v^n + v^{n-1}$ can be eliminated by means of (42). The result becomes

$$u^{n+1} - 2u^n + u^{n-1} = -\Delta t^2\omega^2\frac{1}{4}(u^{n+1} + 2u^n + u^{n-1})\,. \tag{43}$$

It can be shown that

$$\frac{1}{4}(u^{n+1} + 2u^n + u^{n-1}) \approx u^n + \mathcal{O}(\Delta t^2),$$

meaning that (43) is an approximation to the centered scheme (7) for the second-order ODE where the sampling error in the term $\Delta t^2\omega^2 u^n$ is of the same order as the approximation errors in the finite differences, i.e., $\mathcal{O}(\Delta t^2)$. The Crank-Nicolson scheme written as (43) therefore has consistent sampling of all terms at the same time point t_n.

5.4 Comparison of schemes

We can easily compare methods like the ones above (and many more!) with the aid of the Odespy package. Below is a sketch of the code.

```
import odespy
import numpy as np

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

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

    legends = []
```

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

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

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'),
    ]
```

The `vib_undamped_odespy.py` 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 the exact curve $(u(t), v(t))$ is closed and repeats itself for every period. Not all numerical schemes are capable of doing that, meaning that the amplitude instead shrinks or grows with time.

Figure 7 show the results. Note that Odespy applies the label MidpointImplicit for what we have specified as `CrankNicolson` in the code (`CrankNicolson` is just a synonym for class `MidpointImplicit` in the Odespy code). The Forward Euler scheme in Figure 7 has a pronounced spiral curve, pointing to the fact that the amplitude steadily grows, which is also evident in Figure 8. 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 spiral 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.

5.5 Runge-Kutta methods

We may run two other popular standard methods for first-order ODEs, the 2nd- and 4th-order Runge-Kutta methods, to see how they perform. Figures 9 and 10 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, while the 2nd-order scheme suffers from amplitude errors unless the time step is very small.

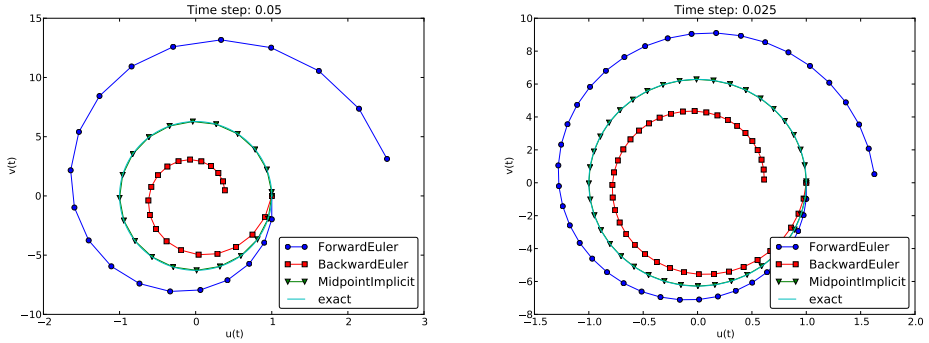


Figure 7: Comparison of classical schemes in the phase plane for two time step values.

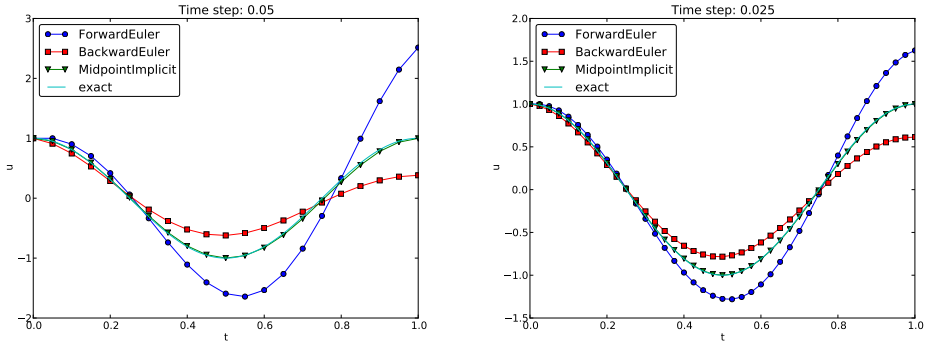


Figure 8: Comparison of solution curves for classical schemes.

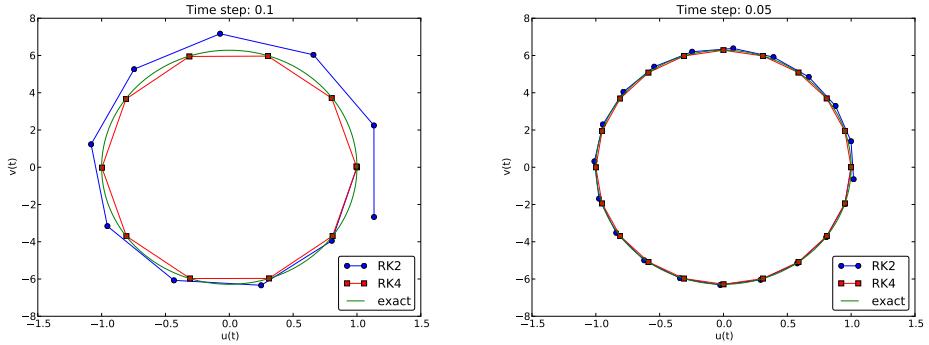


Figure 9: Comparison of Runge-Kutta schemes in the phase plane.

The corresponding results for the Crank-Nicolson scheme are shown in Figure 11. It is clear that the Crank-Nicolson scheme outperforms the 2nd-order Runge-Kutta method. Both schemes have the same order of accuracy $\mathcal{O}(\Delta t^2)$,

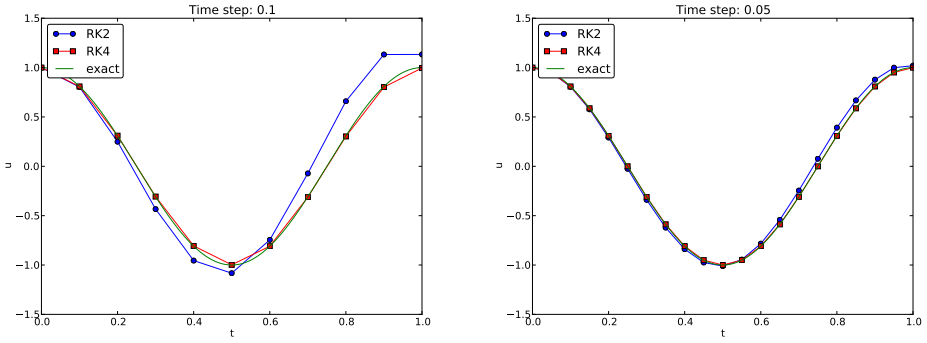


Figure 10: Comparison of Runge-Kutta schemes.

but their differences in the accuracy that matters in a real physical application is very clearly pronounced in this example. Exercise 13 invites you to investigate how the amplitude is computed by a series of famous methods for first-order ODEs.

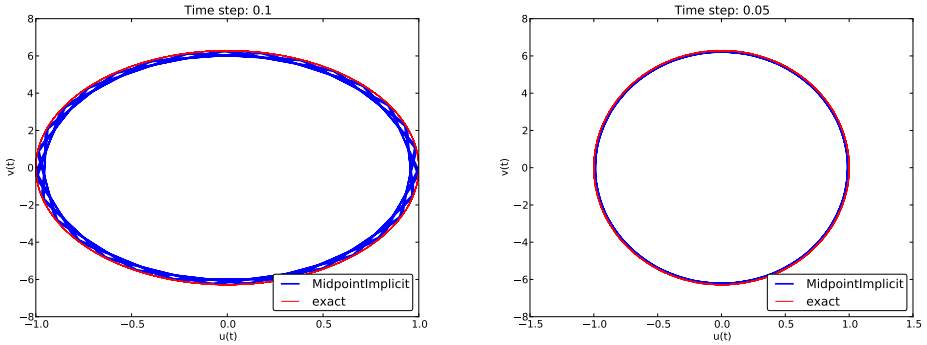


Figure 11: Long-time behavior of the Crank-Nicolson scheme in the phase plane.

5.6 Analysis of the Forward Euler scheme

We may try to find exact solutions of the discrete equations (28)-(29) in the Forward Euler method to better understand why this otherwise useful method has so bad performance for vibration ODEs. An "ansatz" for the solution of the discrete equations is

$$u^n = IA^n,$$
$$v^n = qIA^n,$$

where q and A are scalars to be determined. We could have used a complex exponential form $e^{i\tilde{\omega}n\Delta t}$ since we get oscillatory solutions, but the oscillations grow in the Forward Euler method, so the numerical frequency $\tilde{\omega}$ will be complex anyway (producing an exponentially growing amplitude). Therefore, it is easier to just work with potentially complex A and q as introduced above.

The Forward Euler scheme leads to

$$A = 1 + \Delta t q,$$
$$A = 1 - \Delta t \omega^2 q^{-1}\,.$$

We can easily eliminate A, get $q^2 + \omega^2 = 0$, and solve for

$$q = \pm i\omega,$$

which gives

$$A = 1 \pm \Delta t i\omega\,.$$

We shall take the real part of A^n as the solution. The two values of A are complex conjugates, and the real part of A^n will be the same for both roots. This is easy to realize if we rewrite the complex numbers in polar form, which is also convenient for further analysis and understanding. The polar form $re^{i\theta}$ of a complex number $x + iy$ has $r = \sqrt{x^2 + y^2}$ and $\theta = \tan^{-1}(y/x)$. Hence, the polar form of the two values for A becomes

$$1 \pm \Delta t i\omega = \sqrt{1 + \omega^2\Delta t^2}e^{\pm i\tan^{-1}(\omega\Delta t)}\,.$$

Now it is very easy to compute A^n:

$$(1 \pm \Delta t i\omega)^n = (1 + \omega^2\Delta t^2)^{n/2}e^{\pm ni\tan^{-1}(\omega\Delta t)}\,.$$

Since $\cos(\theta n) = \cos(-\theta n)$, the real parts of the two numbers become the same. We therefore continue with the solution that has the plus sign.

The general solution is $u^n = CA^n$, where C is a constant determined from the initial condition: $u^0 = C = I$. We have $u^n = IA^n$ and $v^n = qIA^n$. The final solutions are just the real part of the expressions in polar form:

$$u^n = I(1 + \omega^2\Delta t^2)^{n/2}\cos(n\tan^{-1}(\omega\Delta t)), \tag{44}$$
$$v^n = -\omega I(1 + \omega^2\Delta t^2)^{n/2}\sin(n\tan^{-1}(\omega\Delta t))\,. \tag{45}$$

The expression $(1 + \omega^2\Delta t^2)^{n/2}$ causes growth of the amplitude, since a number greater than one is raised to a positive exponent $n/2$. We can develop a series expression to better understand the formula for the amplitude. Introducing $p = \omega\Delta t$ as the key variable and using `sympy` gives

```
>>> from sympy import *
>>> p = symbols('p', real=True)
>>> n = symbols('n', integer=True, positive=True)
>>> amplitude = (1 + p**2)**(n/2)
>>> amplitude.series(p, 0, 4)
1 + n*p**2/2 + O(p**4)
```

The amplitude goes like $1 + \frac{1}{2}n\omega^2\Delta t^2$, clearly growing linearly in time (with n).

We can also investigate the error in the angular frequency by a series expansion:

```
>>> n*atan(p).series(p, 0, 4)
n*(p - p**3/3 + O(p**4))
```

This means that the solution for u^n can be written as

$$u^n = (1 + \frac{1}{2}n\omega^2\Delta t^2 + \mathcal{O}(\Delta t^4))\cos\left(\omega t - \frac{1}{3}\omega t\Delta t^2 + \mathcal{O}(\Delta t^4)\right).$$

The error in the angular frequency is of the same order as in the scheme (7) for the second-order ODE, but the error in the amplitude is severe.

6 Energy considerations

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

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

is *constant* for all t. Checking that $E(t)$ really remains constant brings evidence that the numerical computations are sound. It turns out that E is proportional to the mechanical energy in the system. Conservation of energy is much used to check numerical simulations, so it is well invested time to dive into this subject.

6.1 Derivation of the energy expression

We start out with multiplying

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

by u' and integrating from 0 to T:

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

Observing that

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

we get

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

where we have introduced

$$E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2 u^2\,. \tag{46}$$

The important result from this derivation is that the total energy is constant:

$$E(t) = E(0)\,.$$

$E(t)$ is closely related to the system's energy.

The quantity $E(t)$ derived above is physically not the mechanical energy of a vibrating mechanical system, but the energy per unit mass. To see this, we start with Newton's second law $F = ma$ (F is the sum of forces, m is the mass of the system, and a is the acceleration). The displacement u is related to a through $a = u''$. With a spring force as the only force we have $F = -ku$, where k is a spring constant measuring the stiffness of the spring. Newton's 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 equation by finding the work done by each term during a time interval $[0, T]$. To this 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

$$\tilde{E}(t) = E_k(t) + E_p(t) = 0,$$

where

$$E_k(t) = \frac{1}{2}mv^2, \quad v = u', \tag{47}$$

is the *kinetic energy* of the system, and

$$E_p(t) = \frac{1}{2}ku^2 \tag{48}$$

is the *potential energy*. The sum $\tilde{E}(t)$ is the total mechanical energy. The derivation demonstrates the famous energy principle that, under the right physical circumstances, any change in the kinetic energy is due to a change

in potential energy and vice versa. (This principle breaks down when we introduce damping in the system, as we do in Section 10.)

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 $\tilde{E}(t)/m$, i.e., mechanical energy per unit mass.

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

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

Growth of energy in the Forward Euler scheme. It is easy to show that the energy in the Forward Euler scheme increases when stepping from time level n to $n+1$.

$$\begin{aligned} E^{n+1} &= \frac{1}{2}(v^{n+1})^2 + \frac{1}{2}\omega^2(u^{n+1})^2 \\ &= \frac{1}{2}(v^n - \omega^2\Delta t u^n)^2 + \frac{1}{2}\omega^2(u^n + \Delta t v^n)^2 \\ &= (1 + \Delta t^2\omega^2)E^n\,. \end{aligned}$$

6.2 An error measure based on energy

The constant energy is well expressed by its initial value $E(0)$, so that the error in mechanical energy can be computed as a mesh function by

$$e_E^n = \frac{1}{2}\left(\frac{u^{n+1} - u^{n-1}}{2\Delta t}\right)^2 + \frac{1}{2}\omega^2(u^n)^2 - E(0), \quad n = 1, \ldots, N_t - 1, \tag{49}$$

where

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

if $u(0) = I$ and $u'(0) = V$. Note that we have used a centered approximation to u': $u'(t_n) \approx [D_{2t}u]^n$.

A useful norm of the mesh function e_E^n for the discrete mechanical energy can be the maximum absolute value of e_E^n:

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

Alternatively, we can compute other norms involving integration over all mesh points, but we are often interested in worst case deviation of the energy, and then the maximum value is of particular relevance.

A vectorized Python implementation of e_E^n 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 verification. The value of `e_E_norm` is also useful for comparing schemes through their ability to preserve energy. Below is a table demonstrating the relative error in total energy for various schemes (computed by the `vib_undamped_odespy.py` program). The test problem is $u'' + 4\pi^2 u = 0$ with $u(0) = 1$ and $u'(0) = 0$, so the period is 1 and $E(t) \approx 4.93$. We clearly see that the Crank-Nicolson and the Runge-Kutta schemes are superior to the Forward and Backward Euler schemes already after one period.

Method	T	Δt	$\max \lvert e_E^n \rvert / e_E^0$
Forward Euler	1	0.025	$1.678 \cdot 10^{0}$
Backward Euler	1	0.025	$6.235 \cdot 10^{-1}$
Crank-Nicolson	1	0.025	$1.221 \cdot 10^{-2}$
Runge-Kutta 2nd-order	1	0.025	$6.076 \cdot 10^{-3}$
Runge-Kutta 4th-order	1	0.025	$8.214 \cdot 10^{-3}$

However, after 10 periods, the picture is much more dramatic:

Method	T	Δt	$\max \lvert e_E^n \rvert / e_E^0$
Forward Euler	10	0.025	$1.788 \cdot 10^{4}$
Backward Euler	10	0.025	$1.000 \cdot 10^{0}$
Crank-Nicolson	10	0.025	$1.221 \cdot 10^{-2}$
Runge-Kutta 2nd-order	10	0.025	$6.250 \cdot 10^{-2}$
Runge-Kutta 4th-order	10	0.025	$8.288 \cdot 10^{-3}$

The Runge-Kutta and Crank-Nicolson methods hardly change their energy error with T, while the error in the Forward Euler method grows to huge levels and a relative error of 1 in the Backward Euler method points to $E(t) \to 0$ as t grows large.

Running multiple values of Δt, we can get some insight into the convergence of the energy error:

Method	T	Δt	$\max \left\vert e_E^n \right\vert / e_E^0$
Forward Euler	10	0.05	$1.120 \cdot 10^{8}$
Forward Euler	10	0.025	$1.788 \cdot 10^{4}$
Forward Euler	10	0.0125	$1.374 \cdot 10^{2}$
Backward Euler	10	0.05	$1.000 \cdot 10^{0}$
Backward Euler	10	0.025	$1.000 \cdot 10^{0}$
Backward Euler	10	0.0125	$9.928 \cdot 10^{-1}$
Crank-Nicolson	10	0.05	$4.756 \cdot 10^{-2}$
Crank-Nicolson	10	0.025	$1.221 \cdot 10^{-2}$
Crank-Nicolson	10	0.0125	$3.125 \cdot 10^{-3}$
Runge-Kutta 2nd-order	10	0.05	$6.152 \cdot 10^{-1}$
Runge-Kutta 2nd-order	10	0.025	$6.250 \cdot 10^{-2}$
Runge-Kutta 2nd-order	10	0.0125	$7.631 \cdot 10^{-3}$
Runge-Kutta 4th-order	10	0.05	$3.510 \cdot 10^{-2}$
Runge-Kutta 4th-order	10	0.025	$8.288 \cdot 10^{-3}$
Runge-Kutta 4th-order	10	0.0125	$2.058 \cdot 10^{-3}$

A striking fact from this table is that the error of the Forward Euler method is reduced by the same factor as Δt is reduced by, while the error in the Crank-Nicolson method has a reduction proportional to Δt^2 (we cannot say anything for the Backward Euler method). However, for the RK2 method, halving Δt reduces the error by almost a factor of 10 (!), and for the RK4 method the reduction seems proportional to Δt^2 only (and the trend is confirmed by running smaller time steps, so for $\Delta t = 3.9 \cdot 10^{-4}$ the relative error of RK2 is a factor 10 smaller than that of RK4!).

7 The Euler-Cromer method

While the Runge-Kutta methods and the Crank-Nicolson scheme work well for the vibration equation modeled as a first-order ODE system, 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 below. The ideas of the scheme and their further developments have become very popular in particle and rigid body dynamics and hence are widely used by physicists.

7.1 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

$$[D_t^+ u = v]^n, \tag{50}$$

$$[D_t^- v = -\omega^2 u]^{n+1}\,. \tag{51}$$

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

$$u^{n+1} = u^n + \Delta t v^n, \tag{52}$$
$$v^{n+1} = v^n - \Delta t \omega^2 u^{n+1}\,. \tag{53}$$

We realize that after u^{n+1} has been computed from (52), it may be used directly in (53) to compute v^{n+1}.

In physics, it is more common to update the v equation first, with a forward difference, and thereafter the u equation, with a backward difference that applies the most recently computed v value:

$$v^{n+1} = v^n - \Delta t \omega^2 u^n, \tag{54}$$
$$u^{n+1} = u^n + \Delta t v^{n+1}\,. \tag{55}$$

The advantage of ordering the ODEs as in (54)-(55) becomes evident when considering complicated models. Such models are included if we write our vibration ODE more generally as

$$u'' + g(u, u', t) = 0\,.$$

We can rewrite this second-order ODE as two first-order ODEs,

$$v' = -g(u, v, t),$$
$$u' = v\,.$$

This rewrite allows the following scheme to be used:

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

We realize that the first update works well with any g since old values u^n and v^n are used. Switching the equations would demand u^{n+1} and v^{n+1} values in g and result in nonlinear algebraic equations to be solved at each time level.

The scheme (54)-(55) goes under several names: forward-backward scheme, semi-implicit Euler method, semi-explicit Euler, symplectic Euler, Newton-Störmer-Verlet, and Euler-Cromer. We shall stick to the latter name.

How does the Euler-Cromer method preserve the total energy? We may run the example from Section 6.2:

Method	T	Δt	$\max \lvert e_E^n \rvert / e_E^0$
Euler-Cromer	10	0.05	$2.530 \cdot 10^{-2}$
Euler-Cromer	10	0.025	$6.206 \cdot 10^{-3}$
Euler-Cromer	10	0.0125	$1.544 \cdot 10^{-3}$

The relative error in the total energy decreases as Δt^2, and the error level is slightly lower than for the Crank-Nicolson and Runge-Kutta methods.

7.2 Equivalence with the scheme for the second-order ODE

We shall now show that the Euler-Cromer scheme for the system of first-order equations is equivalent to the centered finite difference method for the second-order vibration ODE (!).

We may eliminate the v^n variable from (52)-(53) or (54)-(55). The v^{n+1} term in (54) can be eliminated from (55):

$$u^{n+1} = u^n + \Delta t(v^n - \omega^2 \Delta t u^n)\,. \tag{56}$$

The v^n quantity can be expressed by u^n and u^{n-1} using (55):

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

and when this is inserted in (56) we get

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2 \omega^2 u^n, \tag{57}$$

which is nothing but the centered scheme (7)! The two seemingly different numerical methods are mathematically equivalent. Consequently, the previous analysis of (7) also applies to the Euler-Cromer method. In particular, the amplitude is constant, given that the stability criterion is fulfilled, but there is always an angular frequency error (19). Exercise 18 gives guidance on how to derive the exact discrete solution of the two equations in the Euler-Cromer method.

Although the Euler-Cromer scheme and the method (7) are equivalent, there could be differences in the way they handle the initial conditions. Let us look into this topic. The initial condition $u' = 0$ means $u' = v = 0$. From (54) we get

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

and from (55) it follows that

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

When we previously used a centered approximation of $u'(0) = 0$ combined with the discretization (7) of the second-order ODE, we got a slightly different result: $u^1 = u^0 - \frac{1}{2}\omega^2 \Delta t^2 u^0$. The difference is $\frac{1}{2}\omega^2 \Delta t^2 u^0$, which is of second order in Δt, seemingly consistent with the overall error in the scheme for the differential equation model.

A different view can also be taken. If we approximate $u'(0) = 0$ by a backward difference, $(u^0 - u^{-1})/\Delta t = 0$, we get $u^{-1} = u^0$, and when combined with (7), it results in $u^1 = u^0 - \omega^2 \Delta t^2 u^0$. This means that the Euler-Cromer method based on (55)-(54) corresponds to using only a first-order approximation to the initial condition in the method from Section 1.2.

Correspondingly, using the formulation (52)-(53) with $v^n = 0$ leads to $u^1 = u^0$, which can be interpreted as using a forward difference approximation for the initial condition $u'(0) = 0$. Both Euler-Cromer formulations lead to slightly different values for u^1 compared to the method in Section 1.2. The error is $\frac{1}{2}\omega^2 \Delta t^2 u^0$.

7.3 Implementation

Solver function. The function below, found in `vib_undamped_EulerCromer.py`, implements the Euler-Cromer scheme (54)-(55):

```
import numpy as np

def solver(I, w, dt, T):
    """
    Solve v' = - w**2*u, u'=v for t in (0,T], u(0)=I and v(0)=0,
    by an Euler-Cromer method.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = np.zeros(Nt+1)
    v = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)

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

Verification. Since the Euler-Cromer scheme is equivalent to the finite difference method for the second-order ODE $u'' + \omega^2 u = 0$ (see Section 7.2), the performance of the above `solver` function is the same as for the `solver` function in Section 2. The only difference is the formula for the first time step, as discussed above. This deviation in the Euler-Cromer scheme means that the discrete solution listed in Section 4.4 is not a solution of the Euler-Cromer scheme!

To verify the implementation of the Euler-Cromer method we can adjust `v[1]` so that the computer-generated values can be compared with the formula (20) from in Section 4.4. This adjustment is done in an alternative solver function, `solver_ic_fix` in `vib_EulerCromer.py`. Since we now have an exact solution of the discrete equations available, we can write a test function `test_solver` for checking the equality of computed values with the formula (20):

```
def test_solver():
    """
    Test solver with fixed initial condition against
    equivalent scheme for the 2nd-order ODE u'' + u = 0.
    """
    I = 1.2; w = 2.0; T = 5
    dt = 2/w  # longest possible time step
    u, v, t = solver_ic_fix(I, w, dt, T)
    from vib_undamped import solver as solver2  # 2nd-order ODE
```

```
    u2, t2 = solver2(I, w, dt, T)
    error = np.abs(u - u2).max()
    tol = 1E-14
    assert error < tol
```

Another function, `demo`, visualizes the difference between the Euler-Cromer scheme and the scheme (7) for the second-oder ODE, arising from the mismatch in the first time level.

Using Odespy. The Euler-Cromer method is also available in the Odespy package. The important thing to remember, when using this implementation, is that we must order the unknowns as v and u, so the `u` vector at each time level consists of the velocity v as first component and the displacement u as second component:

```
# Define ODE
def f(u, t, w=1):
    v, u = u
    return [-w**2*u, v]

# Initialize solver
I = 1
w = 2*np.pi
import odespy
solver = odespy.EulerCromer(f, f_kwargs={'w': w})
solver.set_initial_condition([0, I])

# Compute time mesh
P = 2*np.pi/w  # duration of one period
dt = P/timesteps_per_period
Nt = num_periods*timesteps_per_period
T = Nt*dt
import numpy as np
t_mesh = np.linspace(0, T, Nt+1)

# Solve ODE
u, t = solver.solve(t_mesh)
u = u[:,1]  # Extract displacement
```

Convergence rates. We may use the `convergence_rates` function in the file `vib_undamped.py` to investigate the convergence rate of the Euler-Cromer method, see the `convergence_rate` function in the file `vib_undamped_EulerCromer.py`. Since we could eliminate v to get a scheme for u that is equivalent to the finite difference method for the second-order equation in u, we would expect the convergence rates to be the same, i.e., $r = 2$. However, measuring the convergence rate of u in the Euler-Cromer scheme shows that $r = 1$ only! Adjusting the initial condition does not change the rate. Adjusting ω, as outlined in Section 4.2, gives a 4th-order method there, while there is no increase in the measured rate

in the Euler-Cromer scheme. It is obvious that the Euler-Cromer scheme is dramatically much better than the two other first-order methods, Forward Euler and Backward Euler, but this is not reflected in the convergence rate of u.

7.4 The Störmer-Verlet algorithm

Another very popular algorithm for vibration problems, especially for long time simulations, is the Störmer-Verlet algorithm. It has become *the* method among physicists for molecular simulations as well as particle and rigid body dynamics.

The method can be derived by applying the Euler-Cromer idea twice, in a symmetric fashion, during the interval $[t_n, t_{n+1}]$:

1. solve $v' = -\omega u$ by a Forward Euler step in $[t_n, t_{n+\frac{1}{2}}]$
2. solve $u' = v$ by a Backward Euler step in $[t_n, t_{n+\frac{1}{2}}]$
3. solve $u' = v$ by a Forward Euler step in $[t_{n+\frac{1}{2}}, t_{n+1}]$
4. solve $v' = -\omega u$ by a Backward Euler step in $[t_{n+\frac{1}{2}}, t_{n+1}]$

With mathematics,

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

The two steps in the middle can be combined to

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

and consequently

$$v^{n+\frac{1}{2}} = v^n - \frac{1}{2}\Delta t\omega^2 u^n, \tag{58}$$

$$u^{n+1} = u^n + \Delta t v^{n+\frac{1}{2}}, \tag{59}$$

$$v^{n+1} = v^{n+\frac{1}{2}} - \frac{1}{2}\Delta t\omega^2 u^{n+1}\,. \tag{60}$$

Writing the last equation as $v^n = v^{n-\frac{1}{2}} - \frac{1}{2}\Delta t\omega^2 u^n$ and using this v^n in the first equation gives $v^{n+\frac{1}{2}} = v^{n-\frac{1}{2}} - \Delta t\omega^2 u^n$, and the scheme can be written as two steps:

$$v^{n+\frac{1}{2}} = v^{n-\frac{1}{2}} - \Delta t\omega^2 u^n, \tag{61}$$

$$u^{n+1} = u^n + \Delta t v^{n+\frac{1}{2}}, \tag{62}$$

which is nothing but straightforward centered differences for the 2×2 ODE system on a *staggered mesh*, see Section 8.1. We have thus seen that four different reasonings (discretizing $u'' + \omega^2 u$ directly, using Euler-Cromer, using Stömer-Verlet, and using centered differences for the 2×2 system on a staggered mesh) all end up with the same equations! The main difference is that the traditional Euler-Cromer displays first-order convergence in Δt (due to less symmetry in the way u and v are treated) while the others are $\mathcal{O}(\Delta t^2)$ schemes.

The most numerically stable scheme, with respect to accumulation of rounding errors, is (61)-(62). It has, according to [1], better properties in this regard than the direct scheme for the second-order ODE.

8 Staggered mesh

A more intuitive discretization than the Euler-Cromer method, yet equivalent, employs solely centered differences in a natural way for the 2×2 first-order ODE system. The scheme is in fact fully equivalent to the second-order scheme for $u'' + \omega u = 0$, also for the first time step. Such a scheme needs to operate on a *staggered mesh* in time. Staggered meshes are very popular in many physical application, maybe foremost fluid dynamics and electromagnetics, so the topic is important to learn.

8.1 The Euler-Cromer scheme on a staggered mesh

In a staggered mesh, the unknowns are sought at different points in the mesh. Specifically, 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 typically use the notation u^n and $v^{n+\frac{1}{2}}$ for the two unknown mesh functions. Figure 12 presents a graphical sketch of two mesh functions u and v on a staggered mesh.

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

$$[D_t u = v]^{n+\frac{1}{2}}, \tag{63}$$

$$[D_t v = -\omega^2 u]^{n+1}. \tag{64}$$

or if we switch the sequence of the equations:

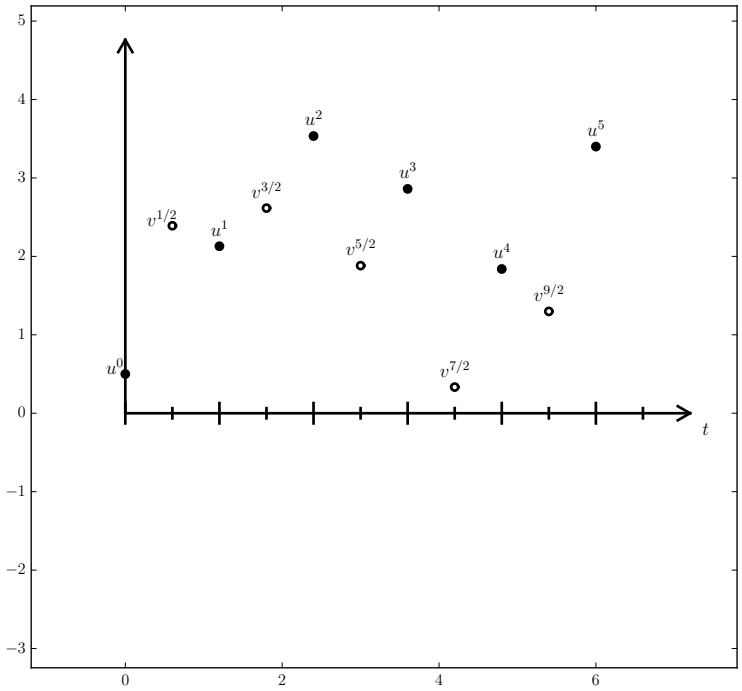


Figure 12: Examples on mesh functions on a staggered mesh in time.

$$[D_t v = -\omega^2 u]^n, \tag{65}$$

$$[D_t u = v]^{n+\frac{1}{2}}\,. \tag{66}$$

Writing out the formulas gives

$$v^{n+\frac{1}{2}} = v^{n-\frac{1}{2}} - \Delta t \omega^2 u^n, \tag{67}$$

$$u^{n+1} = u^n + \Delta t v^{n+\frac{1}{2}}\,. \tag{68}$$

We can eliminate the v values and get back the centered scheme based on the second-order differential equation $u'' + \omega^2 u = 0$, 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 (67) 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 can redo the derivation of the initial conditions when $u'(0) = V$:

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

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

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

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

$$u^0 = I, \tag{69}$$

$$v^{\frac{1}{2}} = V - \frac{1}{2}\Delta t\omega^2 I\,. \tag{70}$$

8.2 Implementation of the scheme on a staggered mesh

The algorithm goes like this:

1. Set the initial values (69) and (70).
2. For $n = 1, 2, \ldots$:
   (a) Compute u^n from (68).
   (b) Compute $v^{n+\frac{1}{2}}$ from (67).

Implementation with integer indices. Translating the schemes (68) and (67) to computer code faces the problem of how to store and access $v^{n+\frac{1}{2}}$, since arrays only allow integer indices with base 0. We must then introduce a convention: $v^{1+\frac{1}{2}}$ is stored in `v[n]` while $v^{1-\frac{1}{2}}$ is stored in `v[n-1]`. We can then write 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
```

Note that u and v are returned together with the mesh points such that the complete mesh function for u is described by `u` and `t`, while `v` and `t_v` represent the mesh function for v.

Implementation with half-integer indices. Some prefer to see a closer relationship between the code and the mathematics for the quantities with half-integer indices. For example, we would like to replace the updating equation 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-half` means `n-1`. A possible solution is to define `half` as a special object such that an integer plus `half` results in the integer, while an integer minus `half` equals the integer 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 add" with `self` as `half` and `other` as `n`). Similarly, the `__rsub__` function is invoked for `n-half` and results in `n-1`.

Using the `half` object, we can implement the algorithms in an even more 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 on
    a staggered mesh with v as unknown at (i+1/2)*dt time points.
    """
    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):
        u[n] = u[n-1] + dt*v[n-half]
        v[n+half] = v[n-half] - dt*w**2*u[n]
    return u, t, v[:-1], t_v[:-1]
```

Verification of this code is easy as we can just compare the computed `u` with the `u` produced by the `solver` function in `vib_undamped.py` (which solves $u'' + \omega^2 u = 0$ directly). The values should coincide to machine precision since the two numerical methods are mathematically equivalent. We refer to the file `vib_undamped_staggered.py` for the details of a unit test (`test_staggered`) that checks this property.

9 Exercises and Problems

Problem 1: Use linear/quadratic functions for verification

Consider the ODE problem

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

a) Discretize this equation according to $[D_t D_t u + \omega^2 u = f]^n$ and derive the equation for the first time step (u^1).

Solution.

For the requested discretization, we get

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

To derive the equation for u^1, we first find the expression for u^{n+1} from the discretized form of the equation. Isolating u^{n+1}, we get

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

With $n = 0$, this expression gives

$$u^1 = \left(2 - (\Delta t\omega)^2\right) u^0 - u^{-1} + \Delta t^2 f^n \,.$$

Here, however, we get a problem with u^{-1}, which appears on the right hand side. To get around that problem, we realize that the initial condition $u' = V$ might be approximated by use of a centered difference approximation as

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

which means that

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

Inserting this expression for u^{-1} into the expression for u^1, we get

$$u^1 = \left(2 - (\Delta t\omega)^2\right) u^0 - (u^1 - 2\Delta t V) + \Delta t^2 f^n \,.$$

Finally, after isolating u^1 on the left hand side, we arrive at

$$u^1 = \left(1 - \frac{1}{2}(\Delta t\omega)^2\right) u^0 + \Delta t V + \frac{1}{2}\Delta t^2 f^n \,.$$

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

Solution.

The initial conditions $u(0) = I$ and $u'(0) = V$ give demands $u_{\mathrm{e}}(0) = I$ and $u'_{\mathrm{e}}(0) = V$, which imply that $d = I$ and c = V.

To compute the source term f, we insert the chosen solution u_{e} into the ODE. This gives

$$0 + \omega^2(ct + d) = f(t),$$

which implies that

$$f(t) = \omega^2(Vt + I) \,.$$

To show that $[D_t D_t t]^n = 0$, we proceed as

$$
\begin{aligned}
[D_tD_t t]^n &= \frac{t^{n+1} - 2t^n + t^{n-1}}{\Delta t^2}, \\
&= \frac{(n+1)\Delta t - 2n\Delta t + (n-1)\Delta t}{\Delta t^2}, \\
&= \frac{n\Delta t + \Delta t - 2n\Delta t + n\Delta t - \Delta t}{\Delta t^2}, \\
&= 0\,.
\end{aligned}
$$

Finally, we show that the chosen u_{e} is also a perfect solution of the discrete equations. If we start by inserting u_{e} into

$$
[D_tD_t u + \omega^2 u = f]^n,
$$

as well as the expression found for f. We get

$$
[D_tD_t(Vt + I) + \omega^2(Vt + I) = \omega^2(Vt + I)]^n,
$$

which can be rewritten as

$$
[D_tD_t(Vt + I)]^n + [\omega^2(Vt + I)]^n = [\omega^2(Vt + I)]^n\,.
$$

Now, since the first term here is zero, we see that the discrete equation is fulfilled exactly for the chosen u_{e} function.

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

```
import sympy as sym
V, t, I, w, dt = sym.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 sym.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 sym.simplify(R)

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

```
    return sym.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), sym.diff(u(t), t).subs(t, 0))

    # Method of manufactured solution requires fitting f
    global f  # source term in the ODE
    f = sym.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()
```

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

Solution.

This part of the code goes as follows:

```
import sympy as sym
import numpy as np

V, t, I, w, dt = sym.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 sym.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 = DtDt(u, dt) + w**2*u(t) - f
    return sym.simplify(R)

def residual_discrete_eq_step1(u):
    """Return the residual of the discrete eq. at the first
    step with u inserted."""
    half = sym.Rational(1,2)
    R = u(t+dt) - I - dt*V - \
        half*dt**2*f.subs(t, 0) + half*dt**2*w**2*I
    R = R.subs(t, 0)  # t=0 in the rhs of the first step eq.
    return sym.simplify(R)

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

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(t)
    print "Initial conditions u(0)=%s, u'(0)=%s:" % \
          (u(t).subs(t, 0), sym.diff(u(t), t).subs(t, 0))

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

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

d) The purpose now is to choose a quadratic function $u_{\mathrm{e}} = bt^2 + ct + d$ as exact solution. Extend the `sympy` code above with a function `quadratic` for fitting `f` and checking if the discrete equations are fulfilled. (The function is very similar to `linear`.)

Solution.

Yes, a quadratic function will fulfill the discrete equations exactly. The implementation becomes

```
def quadratic():
    """Test quadratic function q*t**2 + V*t + I."""
    q = sym.Symbol('q')  # arbitrary constant in t**2 term
    u_e = lambda t: q*t**2 + V*t + I
    main(u_e)
```

Calling `quadratic()` shows that the residual vanishes, and the quadratic function is an exact solution of the discrete equations.

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

Solution.

We can easily make a test:

```
def cubic():
    r, q = sym.symbols('r q')
    main(lambda t: r*t**3 + q*t**2 + V*t + I)
```

When running the final code presented below, the printout shows that the step1 residual for the cubic function is not zero.

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

Solution.

The `solver` function may take the form

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

    u[0] = I
```

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

We can verify the implementation by the following test function:

```
def test_quadratic_exact_solution():
    """Verify solver function via quadratic solution."""
    # Transform global symbolic variables to functions and numbers
    # for numerical computations
    global p, V, I, w
    p, V, I, w = 2.3, 0.9, 1.2, 1.5
    global f, t
    u_e = lambda t: p*t**2 + V*t + I # use p, V, I, w as numbers
    f = ode_source_term(u_e)         # fit source term
    f = sym.lambdify(t, f)           # make function numerical

    dt = 2./w
    u, t = solver(I=I, V=V, f=f, w=w, dt=dt, T=3)
    u_e = u_e(t)
    error = np.abs(u - u_e).max()
    tol = 1E-12
    assert error < tol
    print 'Error in computing a quadratic solution:', error
```

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

Solution.

Here is the complete code for this exercise:

```
import sympy as sym
import numpy as np

V, t, I, w, dt = sym.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 sym.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 = DtDt(u, dt) + w**2*u(t) - f
    return sym.simplify(R)

def residual_discrete_eq_step1(u):
    """Return the residual of the discrete eq. at the first
    step with u inserted."""
    half = sym.Rational(1,2)
    R = u(t+dt) - I - dt*V - \
        half*dt**2*f.subs(t, 0) + half*dt**2*w**2*I
    R = R.subs(t, 0)  # t=0 in the rhs of the first step eq.
    return sym.simplify(R)

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

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(t)
    print "Initial conditions u(0)=%s, u'(0)=%s:" % \
          (u(t).subs(t, 0), sym.diff(u(t), t).subs(t, 0))

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

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


def linear():
    """Test linear function V*t+I: u(0)=I, u'(0)=V."""
    main(lambda t: V*t + I)

def quadratic():
    """Test quadratic function q*t**2 + V*t + I."""
    q = sym.Symbol('q')  # arbitrary constant in t**2 term
    u_e = lambda t: q*t**2 + V*t + I
    main(u_e)
```

```
def cubic():
    r, q = sym.symbols('r q')
    main(lambda t: r*t**3 + q*t**2 + V*t + I)

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

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

def test_quadratic_exact_solution():
    """Verify solver function via quadratic solution."""
    # Transform global symbolic variables to functions and numbers
    # for numerical computations
    global p, V, I, w
    p, V, I, w = 2.3, 0.9, 1.2, 1.5
    global f, t
    u_e = lambda t: p*t**2 + V*t + I # use p, V, I, w as numbers
    f = ode_source_term(u_e)         # fit source term
    f = sym.lambdify(t, f)           # make function numerical

    dt = 2./w
    u, t = solver(I=I, V=V, f=f, w=w, dt=dt, T=3)
    u_e = u_e(t)
    error = np.abs(u - u_e).max()
    tol = 1E-12
    assert error < tol
    print 'Error in computing a quadratic solution:', error

if __name__ == '__main__':
    linear()
    quadratic()
    cubic()
    test_quadratic_exact_solution()
```

Filename: `vib_undamped_verify_mms`.

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 the phase error as the 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.

Solution.

From (19) we have that

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

Dropping the $\mathcal{O}(\Delta t^4)$ term, and since $\omega = \frac{2\pi}{P}$ and $\tilde{\omega} = \frac{2\pi}{\tilde{P}}$, we have that

$$\frac{2\pi}{\tilde{P}} \approx \frac{2\pi}{P}\left(1 + \frac{1}{24}\omega^2\Delta t^2\right)\,.$$

Now, 2π cancels and the remaining equation may be rewritten as

$$P - \tilde{P} \approx \frac{1}{24}\omega^2\Delta t^2\,.$$

This implies that the periods differ by a constant. Since the exact and the numerical solution start out identically, the phase error $P - \tilde{P}$ will become $m\frac{1}{24}\omega^2\Delta t^2$ after m periods, i.e. the phase error is linear in m.

Filename: `vib_phase_error_growth`.

Exercise 3: Improve the accuracy by adjusting the frequency

According to (19), 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 (use $\Delta t = 0.1$ and simulate for 80 periods, with and without adjustment of ω).

Solution.

We may take a copy of the `vib_undamped.py` file and edit the `solver` function to

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

```
def solver(I, w, dt, T, adjust_w=True):
    """
    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)
    if adjust_w:
        w = w*(1 - 1./24*w**2*dt**2)

    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
```

The modified code was run for 80 periods with, and without, the given adjustment of ω. A substantial difference in accuracy was observed between the two, showing that the frequency adjustment improves the situation.

Filename: `vib_adjust_w`.

Exercise 4: 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 software. Use the example from Section 3.2.11 in [2]. 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.

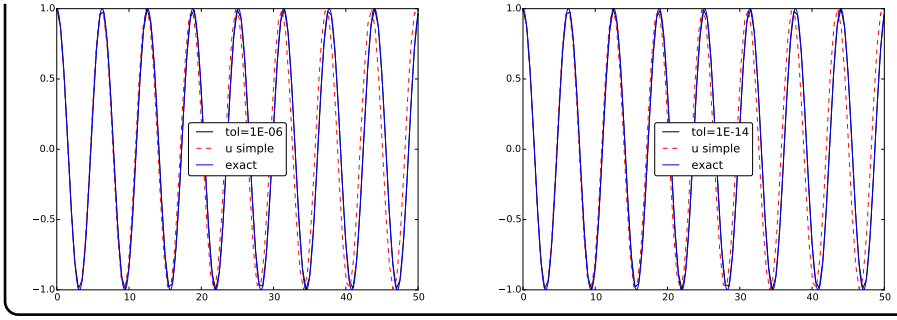
Solution.

Here is a code where we define the test problem, solve it by the Dormand-Prince adaptive method from Odespy, and then call `solver`

```
import odespy
import numpy as np
import sys
#import matplotlib.pyplot as plt
```

```
import scitools.std as plt

def f(s, t):
    u, v = s
    return np.array([v, -u])

def u_exact(t):
    return I*np.cos(w*t)

I = 1; V = 0; u0 = np.array([I, V])
w  = 1; T = 50
tol = float(sys.argv[1])
solver = odespy.DormandPrince(f, atol=tol, rtol=0.1*tol)

Nt = 1 # just one step - let scheme find its intermediate points
t_mesh = np.linspace(0, T, Nt+1)
t_fine = np.linspace(0, T, 10001)

solver.set_initial_condition(u0)
u, t = solver.solve(t_mesh)

# u and t will only consist of [I, u^Nt] and [0,T], i.e. 2 values
# each, while solver.u_all and solver.t_all contain all computed
# points. solver.u_all is a list with arrays, one array (with 2
# values) for each point in time.
u_adaptive = np.array(solver.u_all)

# For comparison, we solve also with simple FDM method
import sys, os
sys.path.insert(0, os.path.join(os.pardir, 'src-vib'))
from vib_undamped import solver as simple_solver
Nt_simple = len(solver.t_all)
dt = float(T)/Nt_simple
u_simple, t_simple = simple_solver(I, w, dt, T)

# Compare in plot: adaptive, constant dt, exact
plt.plot(solver.t_all, u_adaptive[:,0], 'k-')
plt.hold('on')
plt.plot(t_simple, u_simple, 'r--')
plt.plot(t_fine, u_exact(t_fine), 'b-')
plt.legend(['tol=%.0E' % tol, 'u simple', 'exact'])
plt.savefig('tmp_odespy_adaptive.png')
plt.savefig('tmp_odespy_adaptive.pdf')
plt.show()
raw_input()
```

The program may produce the plots seen in the figure below, which shows how the adaptive solution clearly outperforms the simpler method, regardless of the accuracy level.



Filename: `vib_undamped_adaptive`.

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

As an alternative to computing u^1 by (8), one can use a Taylor polynomial with three terms:

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

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.

Solution.

With $u''(0) = -\omega^2 u(0)$ and $u'(0) = 0$, the given Taylor series becomes

$$u(t_1) \approx u(0) + \frac{1}{2}(-\omega^2 u(0))\Delta t^2$$

which may be written as

$$u^1 \approx u^0 - \frac{1}{2}\Delta t^2 \omega^2 u^0$$

but this is nothing but (8).

Now, consider $u'(0) = V$. With a centered difference approximation, this initial condition becomes

$$\frac{u^1 - u^{-1}}{2\Delta t} \approx V$$

which implies that

$$u^{-1} \approx u^1 - 2\Delta t V$$

When $n = 0$, (7) reads

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

Inserting the expression for u^{-1}, we get

$$u^1 = 2u^0 - (u^1 - 2\Delta t V) - \Delta t^2\omega^2 u^0$$

which implies that

$$u^1 = u^0 + \Delta t V - \frac{1}{2}\Delta t^2\omega^2 u^0$$

With the Taylor series approach, we now get

$$u(t_1) \approx u(0) + V\Delta t + \frac{1}{2}(-\omega^2 u(0))\Delta t^2$$

which also gives

$$u^1 = u^0 + \Delta t V - \frac{1}{2}\Delta t^2\omega^2 u^0$$

Filename: `vib_first_step`.

Problem 6: Derive and investigate the velocity Verlet method

The velocity Verlet method for $u'' + \omega^2 u = 0$ is based on the following ideas:

1. step u forward from t_n to t_{n+1} using a three-term Taylor series,
2. replace u'' by $-\omega^2 u$
3. discretize $v' = -\omega^2 u$ by a Crank-Nicolson method.

Derive the scheme, implement it, and determine empirically the convergence rate.

Solution.

Stepping u forward from t_n to t_{n+1} using a three-term Taylor series gives

$$u(t_{n+1}) = u(t_n) + u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2\,.$$

Using $u' = v$ and $u'' = -\omega^2 u$, we get the updating formula

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

Second, the first-order equation for v,

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

is discretized by a centered difference in a Crank-Nicolson fashion at $t_{n+\frac{1}{2}}$:

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

To summarize, we have the scheme

$$u^{n+1} = u^n + v^n \Delta t - \frac{1}{2}\Delta t^2 \omega^2 u^n, \tag{71}$$

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

known as the velocity Verlet algorithm. Observe that this scheme is explicit since u^{n+1} in the second equation is already computed by the first equation.

The algorithm can be straightforwardly implemented as shown below:

```
from vib_undamped import convergence_rates, main

def solver(I, w, dt, T, return_v=False):
    """
    Solve u'=v, v'=-w**2*u for t in (0,T], u(0)=I and v(0)=0,
    by the velocity Verlet method with time step dt.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = np.zeros(Nt+1)
    v = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)

    u[0] = I
    v[0] = 0
    for n in range(Nt):
        u[n+1] = u[n] + v[n]*dt - 0.5*dt**2*w**2*u[n]
        v[n+1] = v[n] - 0.5*dt*w**2*(u[n] + u[n+1])
    if return_v:
        return u, v, t
    else:
        # Return just u and t as in the vib_undamped.py's solver
        return u, t
```

We provide the option that this `solver` function returns the same data as the `solver` function from Section 2.1 (if `return_v` is `False`), but alternatively, it may return `v` along with `u` and `t`.

The error in the Taylor series expansion behind the first equation is $\mathcal{O}(\Delta t^3)$, while the error in the central difference for v is $\mathcal{O}(\Delta t^2)$. The overall

error is then no better than $\mathcal{O}(\Delta t^2)$, which can be verified empirically using the `convergence_rates` function from Section 2.2:

```
>>> import vib_undamped_velocity_Verlet as m
>>> m.convergence_rates(4, solver_function=m.solver)
[2.0036366687367346, 2.0009497328124835, 2.000240105995295]
```

The output confirms that the overall convergence rate is 2.

Problem 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.

Solution.

The smallest period must be $2\Delta t$. Since the period P is related to the angular frequency ω by $P = 2\pi/\omega$, it means that $\omega = \frac{2\pi}{2\Delta t} = \frac{\pi}{\Delta t}$ is the smallest meaningful angular frequency. This further means that the largest value for $\omega\Delta t$ is π.

Filename: `vib_largest_wdt`.

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

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

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`).

Solution.

$$
\begin{aligned}
E &= \frac{[D_t D_t u]^n}{u''(t_n)}, \\
&= \frac{u^{n+1} - 2u^n + u^{n-1}}{u''(t_n)\Delta t^2},
\end{aligned}
$$

Since $u(t) = \exp(i\omega t)$, we have that $u'(t) = i\omega \exp(i\omega t)$ and $u''(t) = (i\omega)^2 \exp(i\omega t) = -\omega^2 \exp(i\omega t)$, so we may proceed with E as

$$
\begin{aligned}
E &= \frac{e^{i\omega(t_n+\Delta t)} - 2e^{i\omega t_n} + e^{i\omega(t_n-\Delta t)}}{-\omega^2 e^{i\omega t_n}\Delta t^2}, \\
&= \frac{e^{i\omega t_n}e^{i\omega\Delta t} - 2e^{i\omega t_n} + e^{i\omega t_n}e^{-i\omega\Delta t}}{-\omega^2 e^{i\omega t_n}\Delta t^2}, \\
&= \frac{e^{i\omega\Delta t} - 2 + e^{-i\omega\Delta t}}{-\omega^2\Delta t^2}, \\
&= \frac{1}{-\omega^2\Delta t^2}\frac{4}{4}\left(e^{i\omega\Delta t} - 2 + e^{-i\omega\Delta t}\right), \\
&= \left(\frac{2}{\omega\Delta t}\right)^2\left(-\frac{e^{i\omega\Delta t} - 2 + e^{-i\omega\Delta t}}{4}\right), \\
&= \left(\frac{2}{\omega\Delta t}\right)^2\left(-\frac{1}{2}\left(\frac{1}{2}e^{i\omega\Delta t} + e^{-i\omega\Delta t} - 1\right)\right), \\
&= \left(\frac{2}{\omega\Delta t}\right)^2\left(-\frac{1}{2}\left(\cos(\omega\Delta t) - 1\right)\right).
\end{aligned}
$$

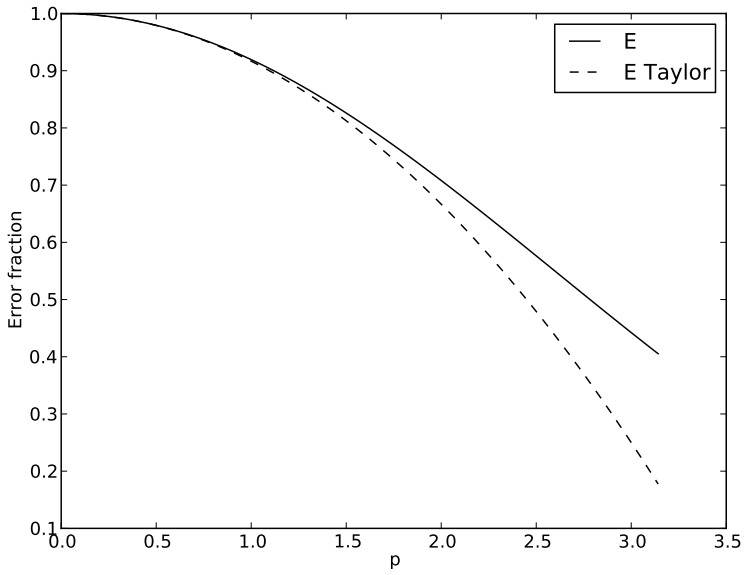
Now, since $\cos(\omega\Delta t) = 1 - 2\sin^2\left(\frac{\omega\Delta t}{2}\right)$, we finally get

$$
\begin{aligned}
E &= \left(\frac{2}{\omega\Delta t}\right)^2\left(-\frac{1}{2}\left(\left(1 - 2\sin^2\left(\frac{\omega\Delta t}{2}\right)\right) - 1\right)\right), \\
&= \left(\frac{2}{\omega\Delta t}\right)^2 \sin^2\left(\frac{\omega\Delta t}{2}\right).
\end{aligned}
$$

```
import matplotlib.pyplot as plt
import numpy as np
import sympy as sym

def E_fraction(p):
    return (2./p)**2*(np.sin(p/2.))**2

a = 0; b = np.pi
```

```
p = np.linspace(a, b, 100)
E_values = np.zeros(len(p))

# create 4th degree Taylor polynomial (also plotted)
p_ = sym.symbols('p_')
E = (2./p_)**2*(sym.sin(p_/2.))**2
E_series = E.series(p_, 0, 4).removeO()
print E_series
E_pyfunc = sym.lambdify([p_], E_series, modules='numpy')

# To avoid division by zero when p is 0, we rather take the limit
E_values[0] = sym.limit(E, p_, 0, dir='+')  # ...when p --> 0, E --> 1
E_values[1:] = E_fraction(p[1:])

plt.plot(p, E_values, 'k-', p, E_pyfunc(p), 'k--')
plt.xlabel('p'); plt.ylabel('Error fraction')
plt.legend(['E', 'E Taylor'])
plt.savefig('tmp_error_fraction.png')
plt.savefig('tmp_error_fraction.pdf')
plt.show()
```

From the plot seen below, we realize how the error fraction E deviates from unity as p grows.



Filename: `vib_plot_fd_exp_error`.

Exercise 9: 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 6.

Make a test function in a separate file, 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, just as for the solution itself.

Solution.

The complete code with test functions goes as follows.

```
import os, sys
sys.path.insert(0, os.path.join(os.pardir, 'src-vib'))
from vib_undamped import solver, u_exact, visualize
import numpy as np

def convergence_rates(m, solver_function, num_periods=8):
    """
    Return m-1 empirical estimates of the convergence rate
    based on m simulations, where the time step is halved
    for each simulation.
    solver_function(I, w, dt, T) solves each problem, where T
    is based on simulation for num_periods periods.
    """
    from math import pi
    w = 0.35; I = 0.3       # just chosen values
    P = 2*pi/w              # period
    dt = P/30               # 30 time step per period 2*pi/w
    T = P*num_periods
    energy_const = 0.5*I**2*w**2    # initial energy when V = 0

    dt_values = []
    E_u_values = []         # error in u
    E_energy_values = []    # error in energy
    for i in range(m):
        u, t = solver_function(I, w, dt, T)
        u_e = u_exact(t, I, w)
        E_u = np.sqrt(dt*np.sum((u_e-u)**2))
        E_u_values.append(E_u)
        energy = 0.5*((u[2:] - u[:-2])/(2*dt))**2 + \
                                    0.5*w**2*u[1:-1]**2
        E_energy = energy - energy_const
        E_energy_norm = np.abs(E_energy).max()
        E_energy_values.append(E_energy_norm)
```

```
        dt_values.append(dt)
        dt = dt/2

    r_u = [np.log(E_u_values[i-1]/E_u_values[i])/
           np.log(dt_values[i-1]/dt_values[i])
           for i in range(1, m, 1)]
    r_E = [np.log(E_energy_values[i-1]/E_energy_values[i])/
           np.log(dt_values[i-1]/dt_values[i])
           for i in range(1, m, 1)]
    return r_u, r_E

def test_convergence_rates():
    r_u, r_E = convergence_rates(
        m=5,
        solver_function=solver,
        num_periods=8)
    # Accept rate to 1 decimal place
    tol = 0.1
    assert abs(r_u[-1] - 2.0) < tol
    assert abs(r_E[-1] - 2.0) < tol

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

Filename: `test_error_conv`.

Exercise 10: Use linear/quadratic functions for verification

This exercise is a generalization of Problem 1 to the extended model problem (75) 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.

Solution.

With a linear spring force, i.e. $s(u) = cu$ (for constant c), our model problem becomes

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

First we consider linear damping, i.e., when $f(u') = bu'$, and follow the text in Section 10.1. Discretizing the equation according to

$$[mD_tD_tu + f(D_{2t}u) + cu = F]^n,$$

implies that

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + b\frac{u^{n+1} - u^{n-1}}{2\Delta t} + cu^n = F^n.$$

The explicit formula for u at each new time level then becomes

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

For the first time step, we use $n = 0$ and a centered difference approximation for the initial condition on the derivative. This gives

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

Next, we consider quadratic damping, i.e., when $f(u') = bu'|u'|$, and follow the text in Chapter 10.2. Discretizing the equation according to

$$[mD_tD_tu + bD_{2t}u|D_{2t}u| + cu = F]^n\,.$$

gives us

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + b\frac{u^{n+1} - u^n}{\Delta t}\frac{|u^n - u^{n-1}|}{\Delta t} + cu^n = F^n.$$

We solve for u^{n+1} to get the explicit updating formula as

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

and the equation for the first time step as

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

Turning to verification with MMS and $u_e(t) = ct + d$, we get $d = I$ and $c = V$ independent of the damping term, so these parameter values stay as for the undamped case.

Proceeding with linear damping, we get from chapter 10.5 that

$$F(t) = bV + c(Vt + I)\,.$$

(Note that there are two different c parameters here, one from $u_e = ct + d$ and one from the spring force cu. The first one disappears, however, as it is switched with V.)

To show that u_e is a perfect solution also to the discrete equations, we insert u_e and F into

$$[mD_tD_tu + bD_{2t}u + cu = F]^n\,.$$

This gives

$$[mD_tD_t(Vt + I) + bD_{2t}(Vt + I) + c(Vt + I) = bV + c(Vt + I)]^n,$$

which may be split up as

$$m[D_tD_t(Vt + I)]^n + b[D_{2t}(Vt + I)]^n + c[(Vt + I)]^n = b[V]^n + c[(Vt + I)]^n.$$

Simplifying, we note that the first term is zero and that $c[(Vt+I)]^n$ appears with the same sign on each side of the equation. Thus, dropping these terms, and cancelling the common factor b, we are left with

$$[D_{2t}(Vt + I)]^n = [V]^n.$$

It therefore remains to show that $[D_{2t}(Vt + I)]^n$ is equal to $[V]^n = V$. We write out the left hand side as

$$\begin{aligned}
[D_{2t}(Vt + I)]^n &= \frac{(Vt_{n+1} + I) - (Vt_{n-1} + I)}{2\Delta t} \\
&= \frac{V(t_{n+1} - t_{n-1})}{2\Delta t} \\
&= \frac{V((t_n + \Delta t) - (t_n - \Delta t))}{2\Delta t} \\
&= V,
\end{aligned}$$

which shows that the two sides of the equation are equal and that the discrete equations are fulfilled exactly for the given u_e function.

If the damping is rather quadratic, we find from chapter 10.5 that

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

As with linear damping, we show that u_e is a perfect solution also to the discrete equations by inserting u_e and F into

$$[mD_tD_tu + bD_{2t}u|D_{2t}u| + cu = F]^n\,.$$

We then get

$$[mD_tD_t(Vt+I)+bD_{2t}(Vt+I)|D_{2t}(Vt+I)|+c(Vt+I) = b|V|V+c(Vt+I)]^n,$$

which simplifies to

$$[bD_{2t}(Vt+I)|D_{2t}(Vt+I)| = b|V|V]^n$$

and further to

$$[D_{2t}(Vt+I)]^n[|D_{2t}(Vt+I)|]^n = |V|V$$

which simply states that

$$V|V| = |V|V\,.$$

Thus, u_e fulfills the discrete equations exactly also when the damping term is quadratic.

When the exact solution is changed to become quadratic or cubic, the situation is more complicated.

For a quadratic solution u_e combined with (zero damping or) linear damping, the output from the program below shows that the discrete equations are fulfilled exactly. However, this is not the case with nonlinear damping, where only the first step gives zero residual.

For a cubic solution u_e, we get a nonzero residual for (zero damping and) linear and nonlinear damping.

```
import sympy as sym
import numpy as np

# The code in vib_undamped_verify_mms.py is here generalized
# to treat the model m*u'' + f(u') + c*u = F(t), where the
# damping term f(u') = 0, b*u' or b*V*abs(V).

def ode_source_term(u, damping):
    """Return the terms in the ODE that the source term
    must balance, here m*u'' + f(u') + c*u.
    u is a symbolic Python function of t."""
    if damping == 'zero':
        return m*sym.diff(u(t), t, t) + c*u(t)
    elif damping == 'linear':
        return m*sym.diff(u(t), t, t) + \
               b*sym.diff(u(t), t) + c*u(t)
    else:  # damping is nonlinear
        return m*sym.diff(u(t), t, t) + \
               b*sym.diff(u(t), t)*abs(sym.diff(u(t), t)) + c*u(t)

def residual_discrete_eq(u, damping):
    """Return the residual of the discrete eq. with u inserted."""
    if damping == 'zero':
        R = m*DtDt(u, dt) + c*u(t) - F
    elif damping == 'linear':
        R = m*DtDt(u, dt) + b*D2t(u, dt) + c*u(t) - F
```

```
    else:   # damping is nonlinear
        R = m*DtDt(u, dt) + b*Dt_p_half(u, dt)*\
            abs(Dt_m_half(u, dt)) + c*u(t) - F
    return sym.simplify(R)

def residual_discrete_eq_step1(u, damping):
    """Return the residual of the discrete eq. at the first
    step with u inserted."""
    half = sym.Rational(1,2)
    if damping == 'zero':
        R = u(t+dt) - u(t) - dt*V - \
            half*dt**2*(F.subs(t, 0)/m) + half*dt**2*(c/m)*I
    elif damping == 'linear':
        R = u(t+dt) - (I + dt*V + \
            half*(dt**2/m)*(-b*V - c*I + F.subs(t, 0)))
    else:   # damping is nonlinear
        R = u(t+dt) - (I + dt*V + \
            half*(dt**2/m)*(-b*V*abs(V) - c*I + F.subs(t, 0)))
    R = R.subs(t, 0)  # t=0 in the rhs of the first step eq.
    return sym.simplify(R)

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

def D2t(u, dt):
    """Return 2nd-order finite difference for u_t.
    u is a symbolic Python function of t.
    """
    return (u(t+dt) - u(t-dt))/(2.0*dt)

def Dt_p_half(u, dt):
    """Return 2nd-order finite difference for u_t, sampled at n+1/2,
    i.e, n pluss one half... u is a symbolic Python function of t.
    """
    return (u(t+dt) - u(t))/dt

def Dt_m_half(u, dt):
    """Return 2nd-order finite difference for u_t, sampled at n-1/2,
    i.e, n minus one half.... u is a symbolic Python function of t.
    """
    return (u(t) - u(t-dt))/dt

def main(u, damping):
    """
    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(t)
    print "Initial conditions u(0)=%s, u'(0)=%s:" % \
          (u(t).subs(t, 0), sym.diff(u(t), t).subs(t, 0))

    # Method of manufactured solution requires fitting F
    global F  # source term in the ODE
    F = sym.simplify(ode_source_term(u, damping))

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


def linear(damping):
    def u_e(t):
        """Return chosen linear exact solution."""
        # General linear function u_e = c*t + d
        # Initial conditions u(0)=I, u'(0)=V require c=V, d=I
        return V*t + I

    main(u_e, damping)

def quadratic(damping):
    # Extend with quadratic functions
    q = sym.Symbol('q')  # arbitrary constant in quadratic term

    def u_e(t):
        return q*t**2 + V*t + I

    main(u_e, damping)

def cubic(damping):
    r, q = sym.symbols('r q')

    main(lambda t: r*t**3 + q*t**2 + V*t + I, damping)


def solver(I, V, F, b, c, m, dt, T, damping):
    """
    Solve m*u'' + f(u') + c*u = F for t in (0,T], u(0)=I and u'(0)=V,
    by a central finite difference method with time step dt.
    F(t) is a callable Python function.
    """
    dt = float(dt)
```

```
    Nt = int(round(T/dt))
    u = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)

    if damping == 'zero':
        u[0] = I
        u[1] = u[0] - 0.5*dt**2*(c/m)*u[0] + \
               0.5*dt**2*F(t[0])/m + dt*V
        for n in range(1, Nt):
            u[n+1] = 2*u[n] - u[n-1] - \
                     dt**2*(c/m)*u[n] + dt**2*F(t[n])/m
    elif damping == 'linear':
        u[0] = I
        u[1] = u[0] + dt*V + \
               0.5*(dt**2/m)*(-b*V - c*u[0] + F(t[0]))
        for n in range(1, Nt):
            u[n+1] = (2*m*u[n] + (b*dt/2.-m)*u[n-1] + \
                     dt**2*(F(t[n])-c*u[n]))/(m+b*dt/2.)
    else:    # damping is quadratic
        u[0] = I
        u[1] = u[0] + dt*V + \
               0.5*(dt**2/m)*(-b*V*abs(V) - c*u[0] + F(t[0]))
        for n in range(1, Nt):
            u[n+1] = 1./(m+b*abs(u[n]-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])-c*u[n]))
    return u, t

def test_quadratic_exact_solution(damping):
    # Transform global symbolic variables to functions and numbers
    # for numerical computations

    global p, V, I, b, c, m
    p, V, I, b, c, m = 2.3, 0.9, 1.2, 2.1, 1.6, 1.3 # i.e., as numbers
    global F, t
    u_e = lambda t: p*t**2 + V*t + I
    F = ode_source_term(u_e, damping) # fit source term
    F = sym.lambdify(t, F)            # ...numerical Python function

    from math import pi, sqrt
    dt = 2*pi/sqrt(c/m)/10   # 10 steps per period 2*pi/w, w=sqrt(c/m)
    u, t = solver(I=I, V=V, F=F, b=b, c=c, m=m, dt=dt,
                  T=(2*pi/sqrt(c/m))*2, damping=damping)
    u_e = u_e(t)
    error = np.abs(u - u_e).max()
    tol = 1E-12
    assert error < tol
    print 'Error in computing a quadratic solution:', error
```

```
if __name__ == '__main__':
    damping = ['zero', 'linear', 'quadratic']
    for e in damping:
        V, t, I, dt, m, b, c = sym.symbols('V t I dt m b c')  # global
        F = None  # global variable for the source term in the ODE
        print '--------------------------------------Damping:', e
        linear(e)   # linear solution used for MMS
        quadratic(e)    # quadratic solution for MMS
        cubic(e)        # ... and cubic
        test_quadratic_exact_solution(e)
```

Filename: `vib_verify_mms`.

Exercise 11: Use an exact discrete solution for verification

Write a test function in a separate file that employs the exact discrete solution (20) to verify the implementation of the `solver` function in the file `vib_undamped.py`.

Solution.

The code goes like this:

```
from vib_undamped import solver
from numpy import arcsin as asin, pi, cos, abs

def test_solver_exact_discrete_solution():
    def tilde_w(w, dt):
        return (2./dt)*asin(w*dt/2.)

    def u_numerical_exact(t):
        return I*cos(tilde_w(w, dt)*t)

    w = 2.5
    I = 1.5

    # Estimate period and time step
    P = 2*pi/w
    num_periods = 4
    T = num_periods*P
    N = 5               # time steps per period
    dt = P/N
    u, t = solver(I, w, dt, T)
    u_e = u_numerical_exact(t)
    error= abs(u_e - u).max()
    # Make a plot in a file, but not on the screen
    from scitools.std import plot
```

```
    plot(t, u, 'bo', t, u_e, 'r-',
         legend=('numerical', 'exact'), show=False,
         savefig='tmp.png')

    assert error < 1E-14

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

Filename: `test_vib_undamped_exact_discrete_sol`.

Exercise 12: Use analytical solution for convergence rate tests

The purpose of this exercise is to perform convergence tests of the problem (75) when $s(u) = cu$, $F(t) = A\sin\phi t$ and there is no damping. Find the complete analytical solution to the problem in this case (most textbooks on mechanics or ordinary differential equations list the various elements you need to write down the exact solution, or you can use symbolic tools like `sympy` or `wolframalpha.com`). 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.

Solution.

The code:

```
import numpy as np
import matplotlib.pyplot as plt
from vib_verify_mms import solver

def u_exact(t, I, V, A, f, c, m):
    """Found by solving mu'' + cu = F in Wolfram alpha."""
    k_1 = I
    k_2 = (V - A*2*np.pi*f/(c - 4*np.pi**2*f**2*m))*\
                                        np.sqrt(m/float(c))
    return A*np.sin(2*np.pi*f*t)/(c - 4*np.pi**2*f**2*m) + \
           k_2*np.sin(np.sqrt(c/float(m))*t) + \
           k_1*np.cos(np.sqrt(c/float(m))*t)

def convergence_rates(N, solver_function, num_periods=8):
    """
    Returns N-1 empirical estimates of the convergence rate
    based on N simulations, where the time step is halved
    for each simulation.
    solver_function(I, V, F, c, m, dt, T, damping) solves
```

```
    each problem, where T is based on simulation for
    num_periods periods.
    """

    def F(t):
        """External driving force"""
        return A*np.sin(2*np.pi*f*t)

    b, c, m = 0, 1.6, 1.3  # just some chosen values
    I = 0                  # init. cond. u(0)
    V = 0                  # init. cond. u'(0)
    A = 1.0                # amplitude of driving force
    f = 1.0                # chosen frequency of driving force
    damping = 'zero'

    P = 1/f
    dt = P/30              # 30 time step per period 2*pi/w
    T = P*num_periods

    dt_values = []
    E_values = []
    for i in range(N):
        u, t = solver_function(I, V, F, b, c, m, dt, T, damping)
        u_e = u_exact(t, I, V, A, f, c, m)
        E = np.sqrt(dt*np.sum((u_e-u)**2))
        dt_values.append(dt)
        E_values.append(E)
        dt = dt/2

    #plt.plot(t, u, 'b--', t, u_e, 'r-'); plt.grid(); plt.show()

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

def test_convergence_rates():
    r = convergence_rates(
        N=5,
        solver_function=solver,
        num_periods=8)
    # Accept rate to 1 decimal place
    tol = 0.1
    assert abs(r[-1] - 2.0) < tol

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

The output from the program shows that r approaches 2.

Filename: `vib_conv_rate`.

Exercise 13: Investigate the amplitude errors of many solvers

Use the program `vib_undamped_odespy.py` from Section 5.4 (utilize the function `amplitudes`) 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 the beginning of Section 5.

Running the code, we get the plots seen in Figure 13, 14, and 15. They show that `RK4` is superior to the others, but that also `CrankNicolson` performs well. In fact, with `RK4` the amplitude changes by less than 0.1 per cent over the interval.

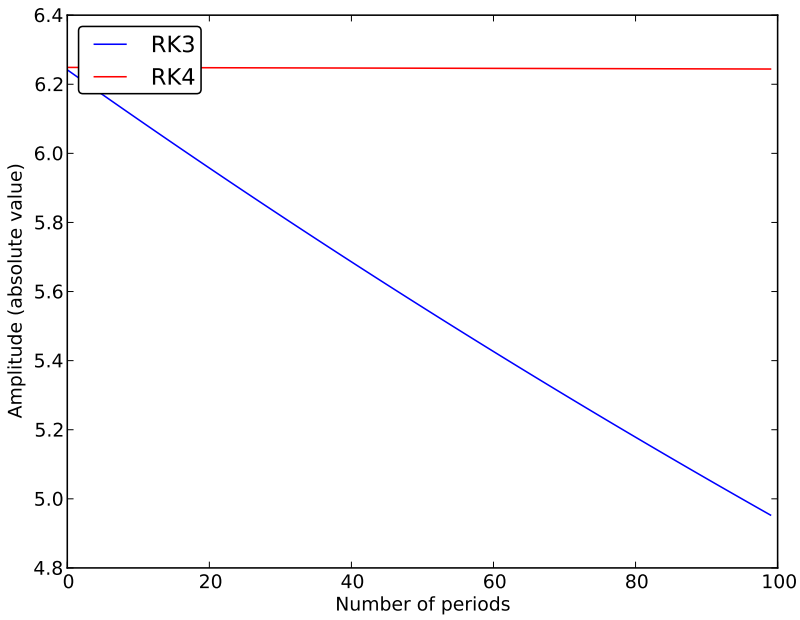


Figure 13: The amplitude as it changes over 100 periods for RK3 and RK4.

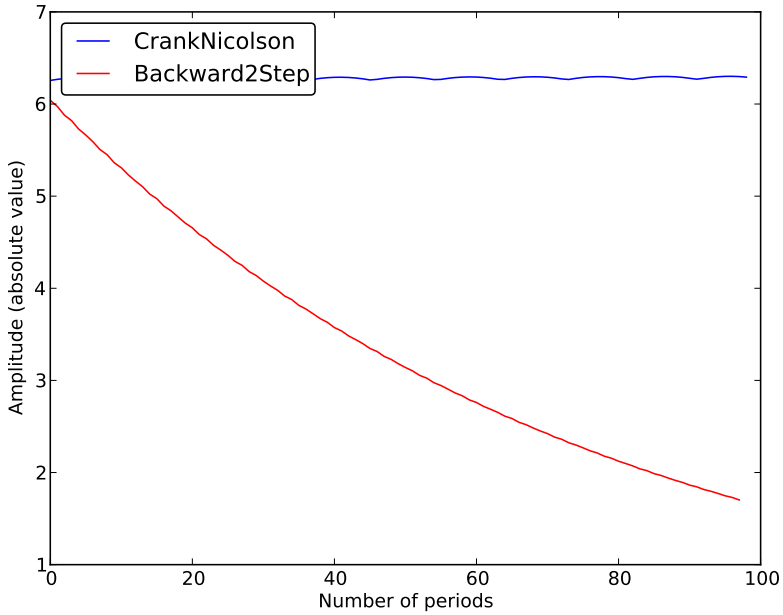


Figure 14: The amplitude as it changes over 100 periods for Crank-Nicolson and Backward 2 step.

Solution.

We modify the proposed code to the following:

```
import scitools.std as plt
#import matplotlib.pyplot as plt
from vib_empirical_analysis import minmax, amplitudes
import sys
import odespy
import numpy as np

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

def run_solvers_and_check_amplitudes(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
```

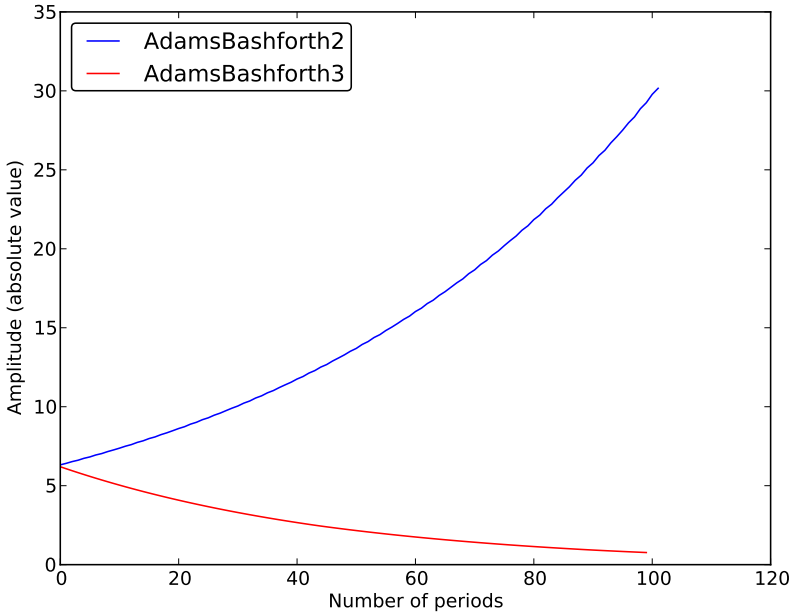


Figure 15: The amplitude as it changes over 100 periods for Adams-Bashforth 2 and 3.

```
Nt = num_periods*timesteps_per_period
T = Nt*dt
t_mesh = np.linspace(0, T, Nt+1)

file_name = 'Amplitudes'   # initialize filename for plot
for solver in solvers:
    solver.set(f_kwargs={'w': w})
    solver.set_initial_condition([0, I])
    u, t = solver.solve(t_mesh)

    solver_name = \
           'CrankNicolson' if solver.__class__.__name__ == \
           'MidpointImplicit' else solver.__class__.__name__
    file_name = file_name + '_' + solver_name

    minima, maxima = minmax(t, u[:,0])
    a = amplitudes(minima, maxima)
    plt.plot(range(len(a)), a, '-', label=solver_name)
    plt.hold('on')
```

```
    plt.xlabel('Number of periods')
    plt.ylabel('Amplitude (absolute value)')
    plt.legend(loc='upper left')
    plt.savefig(file_name + '.png')
    plt.savefig(file_name + '.pdf')
    plt.show()


# Define different sets of experiments
solvers_CNB2 = [odespy.CrankNicolson(f, nonlinear_solver='Newton'),
                odespy.Backward2Step(f)]
solvers_RK34 = [odespy.RK3(f),
                odespy.RK4(f)]
solvers_AB = [odespy.AdamsBashforth2(f),
              odespy.AdamsBashforth3(f)]

if __name__ == '__main__':
    # Default values
    timesteps_per_period = 30
    solver_collection = 'CNB2'
    num_periods = 100
    # Override from command line
    try:
        # Example: python vib_undamped_odespy.py 30 RK34 50
        timesteps_per_period = int(sys.argv[1])
        solver_collection = sys.argv[2]
        num_periods = int(sys.argv[3])
    except IndexError:
        pass # default values are ok
    solvers = eval('solvers_' + solver_collection)  # list of solvers
    run_solvers_and_check_amplitudes(solvers,
                                     timesteps_per_period,
                                     num_periods)
```

Filename: `vib_amplitude_errors`.

Problem 14: Minimize memory usage of a simple vibration solver

We consider the model problem $u'' + \omega^2 u = 0$, $u(0) = I$, $u'(0) = V$, solved by a second-order finite difference scheme. A standard implementation typically employs an array `u` for storing all the u^n values. However, at some time level `n+1` where we want to compute `u[n+1]`, all we need of previous `u` values are from level `n` and `n-1`. We can therefore avoid storing the entire array `u`, and instead work with `u[n+1]`, `u[n]`, and `u[n-1]`, named as `u`, `u_n`, `u_nmp1`, for instance. Another possible naming convention is `u`, `u_n[0]`, `u_n[-1]`. Store the

solution in a file for later visualization. Make a test function that verifies the implementation by comparing with the another code for the same problem.

Solution.

The modified solver function needs more manual steps initially, and it needs shuffling of the `u_n` and `u_nm1` variables at each time level. Otherwise it is very similar to the previous `solver` function with an array `u` for the entire mesh function.

```
import numpy as np
import matplotlib.pyplot as plt

def solver_memsave(I, w, dt, T, filename='tmp.dat'):
    """
    As vib_undamped.solver, but store only the last three
    u values in the implementation. The solution is written to
    file `tmp_memsave.dat`.
    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))
    t = np.linspace(0, Nt*dt, Nt+1)
    outfile = open(filename, 'w')

    u_n = I
    outfile.write('%20.12f %20.12f\n' % (0, u_n))
    u = u_n - 0.5*dt**2*w**2*u_n
    outfile.write('%20.12f %20.12f\n' % (dt, u))
    u_nm1 = u_n
    u_n = u
    for n in range(1, Nt):
        u = 2*u_n - u_nm1 - dt**2*w**2*u_n
        outfile.write('%20.12f %20.12f\n' % (t[n], u))
        u_nm1 = u_n
        u_n = u
    return u, t
```

Verification can be done by comparing with the `solver` function in the `vib_undamped` module. Note that to compare both time series, we need to load the data written to file in `solver_memsave` back in memory again. For this purpose, we can use the `numpy.loadtxt` function, which reads tabular data and returns them as a table `data`. Our interest is in the second column of the data (the `u` values).

```
def test_solver_memsave():
    from vib_undamped import solver
```

```
    _, _ = solver_memsave(I=1, dt=0.1, w=1, T=30)
    u_expected, _ = solver        (I=1, dt=0.1, w=1, T=30)
    data = np.loadtxt('tmp.dat')
    u_computed = data[:,1]
    diff = np.abs(u_expected - u_computed).max()
    assert diff < 5E-13, diff
```

Filename: `vib_memsave0`.

Problem 15: Minimize memory usage of a general vibration solver

The program `vib.py` stores 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 under the names `u`, `u_n`, and `u_nm1` (this is the naming convention used in this book). 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).

Solution.

Here is the complete program:

```
import numpy as np
import scitools.std as plt

def solve_and_store(filename, 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. The solution is written to file (filename).
    Naming convention: we use the name u for the new solution
    to be computed, u_n for the solution one time step prior to
    that and u_nm1 for the solution two time steps prior to that.
    Returns min and max u values needed for subsequent plotting.
    """
    dt = float(dt); b = float(b); m = float(m) # avoid integer div.
    Nt = int(round(T/dt))
    outfile = open(filename, 'w')
    outfile.write('Time          Position\n')
```

```
    u_nm1 = I
    u_min = u_max = u_nm1
    outfile.write('%6.3f         %7.5f\n' % (0*dt, u_nm1))
    if damping == 'linear':
        u_n = u_nm1 + dt*V + dt**2/(2*m)*(-b*V - s(u_nm1) + F(0*dt))
    elif damping == 'quadratic':
        u_n = u_nm1 + dt*V + \
              dt**2/(2*m)*(-b*V*abs(V) - s(u_nm1) + F(0*dt))
    if u_n < u_nm1:
        u_min = u_n
    else:  # either equal or u_n > u_nm1
        u_max = u_n
    outfile.write('%6.3f         %7.5f\n' % (1*dt, u_n))

    for n in range(1, Nt):
        # compute  solution at next time step
        if damping == 'linear':
            u = (2*m*u_n + (b*dt/2 - m)*u_nm1 +
                 dt**2*(F(n*dt) - s(u_n)))/(m + b*dt/2)
        elif damping == 'quadratic':
            u = (2*m*u_n - m*u_nm1 + b*u_n*abs(u_n - u_nm1)
                 + dt**2*(F(n*dt) - s(u_n)))/\
                 (m + b*abs(u_n - u_nm1))
        if u < u_min:
            u_min = u
        elif u > u_max:
            u_max = u

        # write solution to file
        outfile.write('%6.3f         %7.5f\n' % ((n+1)*dt, u))
        # switch references before next step
        u_nm1, u_n, u = u_n, u, u_nm1

    outfile.close()
    return u_min, u_max

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('--b', 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=10)
    parser.add_argument('--window_width', type=float, default=30.,
```

```
                        help='Number of periods in a window')
    parser.add_argument('--damping', type=str, default='linear')
    parser.add_argument('--savefig', action='store_true')
    # Hack to allow --SCITOOLS options
    # (scitools.std reads this argument at import)
    parser.add_argument('--SCITOOLS_easyviz_backend',
                        default='matplotlib')
    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, b, dt, T, window_width, savefig, damping = \
       a.I, a.V, a.m, a.b, a.dt, a.T, a.window_width, a.savefig, \
       a.damping

    filename = 'vibration_sim.dat'
    u_min, u_max = solve_and_store(filename, I, V, m, b, s,
                                   F, dt, T, damping)

    read_and_plot(filename, u_min, u_max)

def read_and_plot(filename, u_min, u_max):
    """
    Read file and plot u vs t line by line in a
    terminal window (only using ascii characters).
    """
    from scitools.avplotter import Plotter
    import time
    umin = 1.2*u_min;  umax = 1.2*u_max
    p = Plotter(ymin=umin, ymax=umax, width=60, symbols='+o')
    fps = 10
    infile = open(filename, 'r')

    # read and treat one line at a time
    infile.readline()   # skip header line
    for line in infile:
        time_and_pos = line.split()  # gives list with 2 elements
        t = float(time_and_pos[0])
        u = float(time_and_pos[1])
        #print 'time: %g   position: %g' % (time, pos)
        print p.plot(t, u), '%.2f' % (t)
        time.sleep(1/float(fps))

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

Filename: `vib_memsave`.

Exercise 16: Implement the Euler-Cromer scheme for the generalized model

We consider the generalized model problem

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

a) Implement the Euler-Cromer method from Section 10.8.

Solution.

A suitable function is

```
import numpy as np
from math import pi

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,
    u'(0)=V by an Euler-Cromer method.
    """
    f = lambda v: b*v if damping == 'linear' else b*abs(v)*v
    dt = float(dt)
    Nt = int(round(T/dt))
    u = np.zeros(Nt+1)
    v = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)

    v[0] = V
    u[0] = I
    for n in range(0, Nt):
        v[n+1] = v[n] + dt*(1./m)*(F(t[n]) - s(u[n]) - f(v[n]))
        u[n+1] = u[n] + dt*v[n+1]
        #print 'F=%g, s=%g, f=%g, v_prev=%g' % (F(t[n]), s(u[n]), f(v[n]), v[n])
        #print 'v[%d]=%g u[%d]=%g' % (n+1,v[n+1],n+1,u[n+1])
    return u, v, t
```

b) We expect the Euler-Cromer method to have first-order convergence rate. Make a unit test based on this expectation.

Solution.

We may use SymPy to derive a problem based on a manufactured solution $u = 3\cos t$. Then we may run some Δt values, compute the error divided by Δt, and check that this ratio remains approximately constant. (An

alternative is to compute true convergence rates and check that they are close to unity.)

```
def test_solver():
    """Check 1st order convergence rate."""
    m = 4; b = 0.1
    s = lambda u: 2*u
    f = lambda v: b*v

    import sympy as sym
    def ode(u):
        """Return source F(t) in ODE for given manufactured u."""
        print 'ode:', m*sym.diff(u, t, 2), f(sym.diff(u,t)), s(u)
        return m*sym.diff(u, t, 2) + f(sym.diff(u,t)) + s(u)

    t = sym.symbols('t')
    u = 3*sym.cos(t)
    F = ode(u)
    F = sym.simplify(F)
    print 'F:', F, 'u:', u
    F = sym.lambdify([t], F, modules='numpy')
    u_exact = sym.lambdify([t], u, modules='numpy')
    I = u_exact(0)
    V = sym.diff(u, t).subs(t, 0)
    print 'V:', V, 'I:', I

    # Numerical parameters
    w = np.sqrt(0.5)
    P = 2*pi/w
    dt_values = [P/20, P/40, P/80, P/160, P/320]
    T = 8*P
    error_vs_dt = []
    for n, dt in enumerate(dt_values):
        u, v, t = solver(I, V, m, b, s, F, dt, T, damping='linear')
        error = np.abs(u - u_exact(t)).max()
        if n > 0:
            error_vs_dt.append(error/dt)
    for i in range(len(error_vs_dt)):
        assert abs(error_vs_dt[i]-
                   error_vs_dt[0]) < 0.1
```

c) Consider a system with $m = 4$, $f(v) = b|v|v$, $b = 0.2$, $s = 2u$, $F = 0$. Compute the solution using the centered difference scheme from Section 10.1 and the Euler-Cromer scheme for the longest possible time step Δt. We can use the result from the case without damping, i.e., the largest $\Delta t = 2/\omega$, $\omega \approx \sqrt{0.5}$ in this case, but since b will modify the frequency, we take the longest possible

time step as a safety factor 0.9 times $2/\omega$. Refine Δt three times by a factor of two and compare the two curves.

Solution.

We rely on the module `vib` for the implementation of the method from Section 10.1. A suitable function for making the comparisons is then

```
def demo():
    """
    Demonstrate difference between Euler-Cromer and the
    scheme for the corresponding 2nd-order ODE.
    """
    I = 1.2; V = 0.2; m = 4; b = 0.2
    s = lambda u: 2*u
    F = lambda t: 0
    w = np.sqrt(2./4)   # approx freq
    dt = 0.9*2/w  # longest possible time step
    w = 0.5
    P = 2*pi/w
    T = 4*P
    from vib import solver as solver2
    import scitools.std as plt
    for k in range(4):
        u2, t2 = solver2(I, V, m, b, s, F, dt, T, 'quadratic')
        u, v, t = solver(I, V, m, b, s, F, dt, T, 'quadratic')
        plt.figure()
        plt.plot(t, u, 'r-', t2, u2, 'b-')
        plt.legend(['Euler-Cromer', 'centered scheme'])
        plt.title('dt=%.3g' % dt)
        raw_input()
        plt.savefig('tmp_%d' % k + '.png')
        plt.savefig('tmp_%d' % k + '.pdf')
        dt /= 2
```

Filename: `vib_EulerCromer`.

Problem 17: 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 of forward and backward differences.

Solution.

$$
\begin{aligned}
[D_t^+ D_t^- u]^n &= [D_t^+ \left(\frac{u^n - u^{n-1}}{\Delta t}\right)]^n \\
&= [\left(\frac{u^{n+1} - u^n}{\Delta t}\right) - \left(\frac{u^n - u^{n-1}}{\Delta t}\right)]^n \\
&= \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} \\
&= [D_t D_t u]^n.
\end{aligned}
$$

Similarly, we get that

$$
\begin{aligned}
[D_t^- D_t^+ u]^n &= [D_t^- \left(\frac{u^{n+1} - u^n}{\Delta t}\right)]^n \\
&= [\left(\frac{u^{n+1} - u^n}{\Delta t}\right) - \left(\frac{u^n - u^{n-1}}{\Delta t}\right)]^n \\
&= \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} \\
&= [D_t D_t u]^n.
\end{aligned}
$$

Filename: `vib_DtDt_fw_bw`.

Exercise 18: Analysis of the Euler-Cromer scheme

The Euler-Cromer scheme for the model problem $u'' + \omega^2 u = 0$, $u(0) = I$, $u'(0) = 0$, is given in (55)-(54). Find the exact discrete solutions of this scheme and show that the solution for u^n coincides with that found in Section 4.

Hint. Use an "ansatz" $u^n = I \exp(i\tilde{\omega}\Delta t\, n)$ and $v^n = qu^n$, where $\tilde{\omega}$ and q are unknown parameters. The following formula is handy:

$$
e^{i\tilde{\omega}\Delta t} + e^{i\tilde{\omega}(-\Delta t)} - 2 = 2\left(\cosh(i\tilde{\omega}\Delta t) - 1\right) = -4\sin^2(\frac{\tilde{\omega}\Delta t}{2}).
$$

Solution.

We follow the ideas in Section 4. Inserting $u^n = I \exp(i\tilde{\omega}\Delta t\, n)$ and $v^n = qu^n$ in (55)-(54) and dividing by $I \exp(i\tilde{\omega}\Delta t\, n)$ gives

$$
q \exp(i\tilde{\omega}\Delta t) = q - \omega^2 \Delta t, \tag{73}
$$

$$
\exp(i\tilde{\omega}\Delta t) = 1 + \Delta t\, q \exp(i\tilde{\omega}\Delta t). \tag{74}
$$

Solving (74) with respect to q gives

$$q = \frac{1}{\Delta t}\left(1 - \exp\left(i\tilde{\omega}\Delta t\right)\right) .$$

Inserting this expression for q in (73) results in

$$\exp\left(i\tilde{\omega}\Delta t\right) + \exp\left(-i\tilde{\omega}\Delta t\right) - 2 = -\omega^2\Delta t^2 .$$

Using the relation $\exp\left(i\tilde{\omega}(\Delta t)\right) + \exp\left(i\tilde{\omega}(-\Delta t)\right) - 2 = -4\sin^2(\frac{\tilde{\omega}\Delta t}{2})$ gives

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

or after dividing by 4,

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

which is the same equation for $\tilde{\omega}$ as found in Section 4, such that $\tilde{\omega}$ is the same. The accuracy, stability, and formula for the exact discrete solution are then all the same as derived in Section 4.

10 Generalization: damping, nonlinearities, and 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 $F(t)$:

$$mu'' + f(u') + s(u) = F(t), \quad u(0) = I,\ u'(0) = V,\ t \in (0, T]\,. \tag{75}$$

We have also included a possibly nonzero initial value for $u'(0)$. 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 $s(u) \sim \sin(u)$.

10.1 A centered scheme for linear damping

Sampling (75) 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

$$[mD_tD_tu + f(D_{2t}u) + s(u) = F]^n, \tag{76}$$

which written out means

$$m\frac{u^{n+1}-2u^n+u^{n-1}}{\Delta t^2}+f(\frac{u^{n+1}-u^{n-1}}{2\Delta t})+s(u^n)=F^n, \tag{77}$$

where F^n as usual means $F(t)$ evaluated at $t=t_n$. Solving (77) 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

$$m\frac{u^{n+1}-2u^n+u^{n-1}}{\Delta t^2}+b\frac{u^{n+1}-u^{n-1}}{2\Delta t}+s(u^n)=F^n, \tag{78}$$

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

$$u^{n+1}=(2mu^n+(\frac{b}{2}\Delta t-m)u^{n-1}+\Delta t^2(F^n-s(u^n)))(m+\frac{b}{2}\Delta t)^{-1}\,. \tag{79}$$

For the first time step we need to discretize $u'(0)=V$ as $[D_{2t}u=V]^0$ and combine with (79) for $n=0$. The discretized initial condition leads to

$$u^{-1}=u^1-2\Delta tV, \tag{80}$$

which inserted in (79) for $n=0$ gives an equation that can be solved for u^1:

$$u^1=u^0+\Delta t\,V+\frac{\Delta t^2}{2m}(-bV-s(u^0)+F^0)\,. \tag{81}$$

10.2 A centered scheme for quadratic damping

When $f(u')=bu'|u'|$, we get a quadratic equation for u^{n+1} in (77). This equation can be straightforwardly solved by the well-known formula for the roots of a quadratic equation. However, we can also avoid the nonlinearity by introducing an approximation with an error of order no higher than what we already have from replacing derivatives with finite differences.

We start with (75) and only replace u'' by D_tD_tu, resulting in

$$[mD_tD_tu+bu'|u'|+s(u)=F]^n\,. \tag{82}$$

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

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

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

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

The next step is to approximate u' at $t_{n\pm 1/2}$, and fortunately a centered difference fits perfectly into the formulas since it involves u values at the mesh points only. With the approximations

$$u'(t_{n+1/2}) \approx [D_t u]^{n+\frac{1}{2}}, \quad u'(t_{n-1/2}) \approx [D_t u]^{n-\frac{1}{2}}, \tag{83}$$

we get

$$[u'|u'|]^n \approx [D_t u]^{n+\frac{1}{2}} |[D_t u]^{n-\frac{1}{2}}| = \frac{u^{n+1}-u^n}{\Delta t}\frac{|u^n - u^{n-1}|}{\Delta t}. \tag{84}$$

The counterpart to (77) is then

$$m\frac{u^{n+1}-2u^n+u^{n-1}}{\Delta t^2} + b\frac{u^{n+1}-u^n}{\Delta t}\frac{|u^n-u^{n-1}|}{\Delta t} + s(u^n) = F^n, \tag{85}$$

which is linear in the unknown u^{n+1}. Therefore, we can easily solve (85) with respect to u^{n+1} and achieve the explicit updating formula

$$\begin{aligned} u^{n+1} = &\left(m + b|u^n - u^{n-1}|\right)^{-1} \times \\ &\left(2mu^n - mu^{n-1} + bu^n|u^n - u^{n-1}| + \Delta t^2(F^n - s(u^n))\right). \end{aligned} \tag{86}$$

In the derivation of a special equation for the first time step we run into some trouble: inserting (80) in (86) for $n = 0$ results in a complicated nonlinear equation for u^1. By thinking differently about the problem we can easily get away with the nonlinearity again. We have for $n = 0$ that $b[u'|u'|]^0 = bV|V|$. Using this value in (82) gives

$$[mD_tD_t u + bV|V| + s(u) = F]^0. \tag{87}$$

Writing this equation out and using (80) results in the special equation for the first time step:

$$u^1 = u^0 + \Delta t V + \frac{\Delta t^2}{2m}\left(-bV|V| - s(u^0) + F^0\right). \tag{88}$$

10.3 A forward-backward discretization of the quadratic damping term

The previous section first proposed to discretize the quadratic damping term $|u'|u'$ using centered differences: $[|D_{2t}|D_{2t}u]^n$. As this gives rise to a nonlinearity in u^{n+1}, it was instead proposed to use a geometric mean combined with centered differences. 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 that the term involves known values, and apply a forward difference to u' to make the term linear in the unknown u^{n+1}. With mathematics,

$$[\beta|u'|u']^n \approx \beta|[D_t^- u]^n|[D_t^+ u]^n = \beta\left|\frac{u^n - u^{n-1}}{\Delta t}\right|\frac{u^{n+1} - u^n}{\Delta t}. \tag{89}$$

The forward and backward differences both have an error proportional to Δt so one may think the discretization above leads to a first-order scheme. However, by looking at the formulas, we realize that the forward-backward differences in (89) result in exactly the same scheme as in (85) where we used a geometric mean and centered differences and committed errors of size $\mathcal{O}(\Delta t^2)$. Therefore, the forward-backward differences in (89) act in a symmetric way and actually produce a second-order accurate discretization of the quadratic damping term.

10.4 Implementation

The algorithm arising from the methods in Sections 10.1 and 10.2 is very similar to the undamped case in Section 1.2. The difference is basically a question of different formulas for u^1 and u^{n+1}. This is actually quite remarkable. The equation (75) is normally impossible to solve by pen and paper, but possible for some special choices of F, s, and f. On the contrary, the complexity of the nonlinear generalized model (75) versus the simple undamped model is not a big deal when we solve the problem numerically!

The computational algorithm takes the form

1. $u^0 = I$
2. compute u^1 from (81) if linear damping or (88) if quadratic damping
3. for $n = 1, 2, \ldots, N_t - 1$:
   (a) compute u^{n+1} from (79) if linear damping or (86) 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 = np.zeros(Nt+1)
    t = np.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
```

The complete code resides in the file `vib.py`.

10.5 Verification

Constant solution. For debugging and initial verification, a constant solution is often very useful. We choose $u_{\mathrm{e}}(t) = I$, which implies $V = 0$. Inserted in the ODE, we get $F(t) = s(I)$ for any choice of f. Since the discrete derivative of a constant vanishes (in particular, $[D_{2t}I]^n = 0$, $[D_tI]^n = 0$, and $[D_tD_tI]^n = 0$), the constant solution also fulfills the discrete equations. The constant should therefore be reproduced to machine precision. The function `test_constant` in `vib.py` implements this test.

Linear solution. Now we choose a linear solution: $u_{\mathrm{e}} = ct + d$. The initial condition $u(0) = I$ implies $d = I$, and $u'(0) = V$ forces c to be V. Inserting $u_{\mathrm{e}} = Vt + I$ in the ODE with linear damping results in

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

while quadratic damping requires the source term

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

Since the finite difference approximations used to compute u' all are exact for a linear function, it turns out that the linear u_{e} is also a solution of the discrete equations. Exercise 10 asks you to carry out all the details.

Quadratic solution. Choosing $u_{\mathrm{e}} = bt^2 + Vt + I$, with b arbitrary, fulfills the initial conditions and fits the ODE if F is adjusted properly. The solution also solves the discrete equations with linear damping. However, this quadratic polynomial in t does not fulfill the discrete equations in case of quadratic damping, because the geometric mean used in the approximation of this term introduces an error. Doing Exercise 10 will reveal the details. One can fit F^n in the discrete

equations such that the quadratic polynomial is reproduced by the numerical method (to machine precision).

Catching bugs. How good are the constant and quadratic solutions at catching bugs in the implementation? Let us check that by introducing some bugs.

- Use `m` instead of `2*m` in the denominator of `u[1]`: code works for constant solution, but fails (as it should) for a quadratic one.
- Use `b*dt` instead of `b*dt/2` in the updating formula for `u[n+1]` in case of linear damping: constant and quadratic both fail.
- Use `F[n+1]` instead of `F[n]` in case of linear or quadratic damping: constant solution works, quadratic fails.

We realize that the constant solution is very useful for catching certain bugs because of its simplicity (easy to predict what the different terms in the formula should evaluate to), while the quadratic solution seems capable of detecting all (?) other kinds of typos in the scheme. These results demonstrate why we focus so much on exact, simple polynomial solutions of the numerical schemes in these writings.

10.6 Visualization

The functions for visualizations differ significantly from those in the undamped case in the `vib_undamped.py` program because, in the present general case, we 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 to get a plot that moves with the graph and shows the most recent parts of it in long time simulations.

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

10.7 User interface

The `main` function is changed substantially 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 (75). 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

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

This results in a moving window following the function on the screen. Figure 16 shows a part of the time series.

10.8 The Euler-Cromer scheme for the generalized model

The ideas of the Euler-Cromer method from Section 7 carry over to the generalized model. We write (75) as two equations for u and $v = u'$. The first equation is taken as the one with v' on the left-hand side:

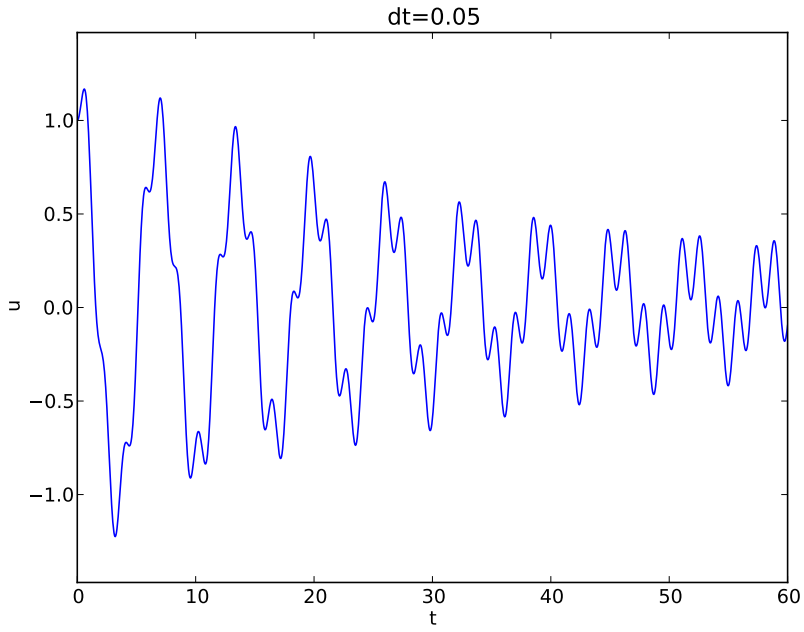


Figure 16: Damped oscillator excited by a sinusoidal function.

$$v' = \frac{1}{m}(F(t) - s(u) - f(v)), \tag{90}$$

$$u' = v\,. \tag{91}$$

Again, the idea is to step (90) forward using a standard Forward Euler method, while we update u from (91) with a Backward Euler method, utilizing the recent, computed v^{n+1} value. In detail,

$$\frac{v^{n+1} - v^n}{\Delta t} = \frac{1}{m}(F(t_n) - s(u^n) - f(v^n)), \tag{92}$$

$$\frac{u^{n+1} - u^n}{\Delta t} = v^{n+1}, \tag{93}$$

resulting in the explicit scheme

$$v^{n+1} = v^n + \Delta t \frac{1}{m}(F(t_n) - s(u^n) - f(v^n)), \tag{94}$$

$$u^{n+1} = u^n + \Delta t\, v^{n+1}\,. \tag{95}$$

We immediately note one very favorable feature of this scheme: all the nonlinearities in $s(u)$ and $f(v)$ are evaluated at a previous time level. This makes the Euler-Cromer method easier to apply and hence much more convenient than the centered scheme for the second-order ODE (75).

The initial conditions are trivially set as

$$v^0 = V, \tag{96}$$
$$u^0 = I\,. \tag{97}$$

10.9 The Störmer-Verlet algorithm for the generalized model

We can easily apply the ideas from Section 7.4 to extend that method to the generalized model

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

However, since the scheme is essentially centered differences for the ODE system on a staggered mesh, we do not go into detail here, but refer to Section 10.10.

10.10 A staggered Euler-Cromer scheme for a generalized model

The more general model for vibration problems,

$$mu'' + f(u') + s(u) = F(t), \quad u(0) = I,\ u'(0) = V,\ t \in (0, T], \tag{98}$$

can be rewritten as a first-order ODE system

$$v' = m^{-1}\left(F(t) - f(v) - s(u)\right), \tag{99}$$
$$u' = v\,. \tag{100}$$

It is natural to introduce a staggered mesh (see Section 8.1) and seek u at mesh points t_n (the numerical value is denoted by u^n) and v between mesh points at $t_{n+1/2}$ (the numerical value is denoted by $v^{n+\frac{1}{2}}$). A centered difference approximation to (100)-(99) can then be written in operator notation as

$$[D_t v = m^{-1}\left(F(t) - f(v) - s(u)\right)]^n, \tag{101}$$
$$[D_t u = v]^{n+\frac{1}{2}}\,. \tag{102}$$

Written out,

$$\frac{v^{n+\frac{1}{2}} - v^{n-\frac{1}{2}}}{\Delta t} = m^{-1}\left(F^n - f(v^n) - s(u^n)\right), \tag{103}$$

$$\frac{u^n - u^{n-1}}{\Delta t} = v^{n+\frac{1}{2}}. \tag{104}$$

With linear damping, $f(v) = bv$, we can use an arithmetic mean for $f(v^n)$: $f(v^n) \approx= \frac{1}{2}(f(v^{n-\frac{1}{2}}) + f(v^{n+\frac{1}{2}}))$. The system (103)-(104) can then be solved with respect to the unknowns u^n and $v^{n+\frac{1}{2}}$:

$$v^{n+\frac{1}{2}} = \left(1 + \frac{b}{2m}\Delta t\right)^{-1}\left(v^{n-\frac{1}{2}} + \Delta t m^{-1}\left(F^n - \frac{1}{2}f(v^{n-\frac{1}{2}}) - s(u^n)\right)\right), \tag{105}$$

$$u^n = u^{n-1} + \Delta t v^{n-\frac{1}{2}}. \tag{106}$$

In case of quadratic damping, $f(v) = b|v|v$, we can use a geometric mean: $f(v^n) \approx b|v^{n-\frac{1}{2}}|v^{n+\frac{1}{2}}$. Inserting this approximation in (103)-(104) and solving for the unknowns u^n and $v^{n+\frac{1}{2}}$ results in

$$v^{n+\frac{1}{2}} = (1 + \frac{b}{m}|v^{n-\frac{1}{2}}|\Delta t)^{-1}\left(v^{n-\frac{1}{2}} + \Delta t m^{-1}\left(F^n - s(u^n)\right)\right), \tag{107}$$

$$u^n = u^{n-1} + \Delta t v^{n-\frac{1}{2}}. \tag{108}$$

The initial conditions are derived at the end of Section 8.1:

$$u^0 = I, \tag{109}$$

$$v^{\frac{1}{2}} = V - \frac{1}{2}\Delta t\omega^2 I. \tag{110}$$

10.11 The PEFRL 4th-order accurate algorithm

A variant of the Euler-Cromer type of algorithm, which provides an error $\mathcal{O}(\Delta t^4)$ if $f(v) = 0$, is called PEFRL [4]. This algorithm is very well suited for integrating dynamic systems (especially those without damping) over very long time periods. Define

$$g(u, v) = \frac{1}{m}(F(t) - s(u) - f(v)).$$

The algorithm is explicit and features these steps:

$$u^{n+1,1} = u^n + \xi\Delta t v^n, \tag{111}$$
$$v^{n+1,1} = v^n + \frac{1}{2}(1-2\lambda)\Delta t g(u^{n+1,1}, v^n), \tag{112}$$
$$u^{n+1,2} = u^{n+1,1} + \chi\Delta t v^{n+1,1}, \tag{113}$$
$$v^{n+1,2} = v^{n+1,1} + \lambda\Delta t g(u^{n+1,2}, v^{n+1,1}), \tag{114}$$
$$u^{n+1,3} = u^{n+1,2} + (1-2(\chi+\xi))\Delta t v^{n+1,2}, \tag{115}$$
$$v^{n+1,3} = v^{n+1,2} + \lambda\Delta t g(u^{n+1,3}, v^{n+1,2}), \tag{116}$$
$$u^{n+1,4} = u^{n+1,3} + \chi\Delta t v^{n+1,3}, \tag{117}$$
$$v^{n+1} = v^{n+1,3} + \frac{1}{2}(1-2\lambda)\Delta t g(u^{n+1,4}, v^{n+1,3}), \tag{118}$$
$$u^{n+1} = u^{n+1,4} + \xi\Delta t v^{n+1} \tag{119}$$

The parameters ξ, λ, and ξ have the values

$$\xi = 0.1786178958448091, \tag{120}$$
$$\lambda = -0.2123418310626054, \tag{121}$$
$$\chi = -0.06626458266981849 \tag{122}$$

11 Exercises and Problems

Exercise 19: 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 Sections 5.5.1 and 5.5.2 in [2]. 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.

Solution.

The complete code looks like this.

```
# Reimplementation of vib.py using classes

import numpy as np
import scitools.std as plt
import sympy as sym
from vib import solver as vib_solver
```

```
from vib import visualize as vib_visualize
from vib import visualize_front as vib_visualize_front
from vib import visualize_front_ascii as vib_visualize_front_ascii
from vib import plot_empirical_freq_and_amplitude as \
                           vib_plot_empirical_freq_and_amplitude

class Vibration:
    '''
    Problem: m*u'' + f(u') + s(u) = F(t) for t in (0,T],
    u(0)=I and u'(0)=V. The problem is solved
    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'). Zero damping is achieved
    with b=0. F(t) and s(u) are Python functions.
    '''
    def __init__(self, I=1, V=0, m=1, b=0, damping='linear'):
        self.I = I; self.V = V; self.m = m; self.b=b;
        self.damping = damping
    def s(self, u):
        return u
    def F(self, t):
        '''Driving force. Zero implies free oscillations'''
        return 0

class Free_vibrations(Vibration):
    '''F(t) = 0'''
    def __init__(self, s=None, I=1, V=0, m=1, b=0, damping='linear'):
        Vibration.__init__(self, I=I, V=V, m=m, b=b, damping=damping)
        if s != None:
            self.s = s

class Forced_vibrations(Vibration):
    '''F(t)! = 0'''
    def __init__(self, F, s=None, I=1, V=0, m=1, b=0,
                 damping='linear'):
        Vibration.__init__(self, I=I, V=V, m=m, b=b,
                           damping=damping)
        if s != None:
            self.s = s
        self.F = F

class Solver:
    def __init__(self, dt=0.05, T=20):
        self.dt = dt; self.T = T

    def solve(self, problem):
        self.u, self.t = vib_solver(
            problem.I, problem.V,
```

```
            problem.m, problem.b,
            problem.s, problem.F,
            self.dt, self.T, problem.damping)

class Visualizer:
    def __init__(self, problem, solver, window_width, savefig):
        self.problem = problem; self.solver = solver
        self.window_width = window_width; self.savefig = savefig
    def visualize(self):
        u = self.solver.u; t = self.solver.t   # short forms
        num_periods = vib_plot_empirical_freq_and_amplitude(u, t)
        if num_periods <= 40:
            plt.figure()
            vib_visualize(u, t)
        else:
            vib_visualize_front(u, t, self.window_width, self.savefig)
            vib_visualize_front_ascii(u, t)
        plt.show()

def main():
    # Note: the reading of parameter values would better be done
    # from each relevant class, i.e. class Problem should read I, V,
    #  etc., while class Solver should read dt and T, and so on.
    # Consult, e.g., Langtangen: "A Primer on Scientific Programming",
    # App E.
    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('--b', type=float, default=0.0)
    parser.add_argument('--s', type=str, default=None)
    parser.add_argument('--F', type=str, default='0')
    parser.add_argument('--dt', type=float, default=0.05)
    parser.add_argument('--T', type=float, default=20)
    parser.add_argument('--window_width', type=float, default=30.,
                        help='Number of periods in a window')
    parser.add_argument('--damping', type=str, default='linear')
    parser.add_argument('--savefig', action='store_true')
    # Hack to allow --SCITOOLS options
    # (scitools.std reads this argument at import)
    parser.add_argument('--SCITOOLS_easyviz_backend',
                        default='matplotlib')
    a = parser.parse_args()

    from scitools.std import StringFunction
    if a.s != None:
        s = StringFunction(a.s, independent_variable='u')
```

```
    else:
        s = None
    F = StringFunction(a.F, independent_variable='t')

    if a.F == '0':  # free vibrations
        problem = Free_vibrations(s=s, I=a.I, V=a.V, m=a.m, b=a.b,
                                  damping=a.damping)
    else:   # forced vibrations
        problem = Forced_vibrations(lambda t: np.sin(t),
                                    s=s, I=a.I, V=a.V,
                                    m=a.m, b=a.b, damping=a.damping)

    solver = Solver(dt=a.dt, T=a.T)
    solver.solve(problem)

    visualizer = Visualizer(problem, solver,
                            a.window_width, a.savefig)
    visualizer.visualize()

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

Filename: `vib_class`.

Problem 20: 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` 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 launched in this exercise. Perform numerical experiments to investigate how much accuracy that is lost by using the backward differences.

Solution.

The new discretization approach of the linear and quadratic damping terms calls for new derivations of the updating formulas (for u) in the solver. Since backward difference approximations will be used for the damping term, we may also use this approximation for the initial condition on $u'(0)$ without deteriorating the convergence rate any further. Note that introducing backward difference approximations for the damping term make our computational schemes first order, as opposed to the original second order schemes which used central difference approximations also for the damping terms. The motivation for also using a backward difference approximation for the initial condition on $u'(0)$, is simply that the computational schemes get much simpler.

With linear damping, the new discretized form of the equation reads

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

which gives us

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

With $n=0$, the updating formula becomes

$$u^1=\left(2-\frac{\Delta tb}{m}\right)u^0+\frac{\Delta t^2}{m}\left(F^0-s(u^0)\right)+\left(\frac{\Delta tb}{m}-1\right)u^{-1},$$

which requires some further elaboration because of the unknown u^{-1}. We handle this by discretizing the initial condition $u'(0)=V$ by a backward difference approximation as

$$\frac{u^0-u^{-1}}{\Delta t}=V,$$

which implies that

$$u^{-1}=u^0-\Delta tV.$$

Inserting this expression for u^{-1} in the updating formula for u^{n+1}, and simplifying, gives us the following special formula for the first time step:

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

Switching to quadratic damping, the new discretized form of the equations becomes

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

which leads to

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

With $n=0$, this updating formula becomes

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

Again, we handle the unknown u^{-1} via the same expression as above, which be derived from a backward difference approximation to the initial condition on the derivative. Inserting this expression for u^{-1} and simplifying, gives the special updating formula for u^1 as

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

We implement these new computational schemes in a new solver function `solver_bwdamping`, so that the discrete solution for u can be found by both the original and the new solver. The difference between the two different solutions is then visualized in the same way as the original solution in `main`.

The convergence rates computed in `test_mms` demonstrates that our scheme now is a first order scheme, as r is seen to approach 1.0 with decreasing Δt.

Both solvers reproduce a constant solution exactly (within machine precision), whereas sinusoidal and quadratic solutions differ, as should be expected after comparing the schemes. Pointing out the "best" approach is difficult: the backward differences yield a much simpler mathematical problem to be solved, while the more complicated method converges faster and gives more accuracy for the same cost. On the other hand, the backward differences can yield any reasonable accuracy by lowering Δt, and the results are obtained within a few seconds on a laptop.

Here is the complete computer code, arising from copying `vib.py` and changing the functions that have to be changed:

```
import numpy as np
#import matplotlib.pyplot as plt
import scitools.std as plt

def solver_bwdamping(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. All terms except damping is discretized
    by a central finite difference method with time step dt.
    The damping term is discretized by a backward diff. approx.,
    as is the init.cond. u'(0). 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 = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)

    u_original = np.zeros(Nt+1); u_original[0] = I  # for testing

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


import sympy as sym

def test_constant():
    """Verify a constant solution."""
    u_exact = lambda t: I
    I = 1.2; V = 0; m = 2; b = 0.9
    w = 1.5
    s = lambda u: w**2*u
    F = lambda t: w**2*u_exact(t)
    dt = 0.2
    T = 2
    #u, t = solver(I, V, m, b, s, F, dt, T, 'linear')
    u, t = solver_bwdamping(I, V, m, b, s, F, dt, T, 'linear')
    difference = np.abs(u_exact(t) - u).max()
    print difference
    tol = 1E-13
```

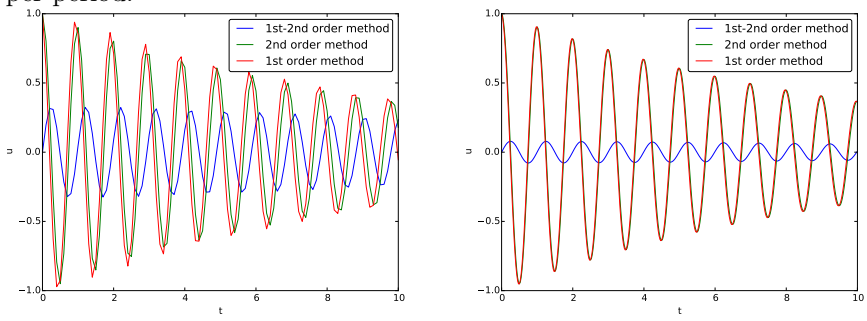
```
    assert difference < tol

    #u, t = solver(I, V, m, b, s, F, dt, T, 'quadratic')
    u, t = solver_bwdamping(I, V, m, b, s, F, dt, T, 'quadratic')
    difference = np.abs(u_exact(t) - u).max()
    print difference
    assert difference < tol

def lhs_eq(t, m, b, s, u, damping='linear'):
    """Return lhs of differential equation as sympy expression."""
    v = sym.diff(u, t)
    if damping == 'linear':
        return m*sym.diff(u, t, t) + b*v + s(u)
    else:
        return m*sym.diff(u, t, t) + b*v*sym.Abs(v) + s(u)

def test_quadratic():
    """Verify a quadratic solution."""
    I = 1.2; V = 3; m = 2; b = 0.9
    s = lambda u: 4*u
    t = sym.Symbol('t')
    dt = 0.2
    T = 2

    q = 2  # arbitrary constant
    u_exact = I + V*t + q*t**2
    F = sym.lambdify(t, lhs_eq(t, m, b, s, u_exact, 'linear'))
    u_exact = sym.lambdify(t, u_exact, modules='numpy')
    #u1, t1 = solver(I, V, m, b, s, F, dt, T, 'linear')
    u1, t1 = solver_bwdamping(I, V, m, b, s, F, dt, T, 'linear')
    diff = np.abs(u_exact(t1) - u1).max()
    print diff
    tol = 1E-13
    #assert diff < tol

    # In the quadratic damping case, u_exact must be linear
    # in order to exactly recover this solution
    u_exact = I + V*t
    F = sym.lambdify(t, lhs_eq(t, m, b, s, u_exact, 'quadratic'))
    u_exact = sym.lambdify(t, u_exact, modules='numpy')
    #u2, t2 = solver(I, V, m, b, s, F, dt, T, 'quadratic')
    u2, t2 = solver_bwdamping(I, V, m, b, s, F, dt, T, 'quadratic')
    diff = np.abs(u_exact(t2) - u2).max()
    print diff
    assert diff < tol

def test_sinusoidal():
    """Verify a numerically exact sinusoidal solution when b=F=0."""
```

```
    from math import asin

    def u_exact(t):
        w_numerical = 2/dt*np.arcsin(w*dt/2)
        return I*np.cos(w_numerical*t)

    I = 1.2; V = 0; m = 2; b = 0
    w = 1.5  # fix the frequency
    s = lambda u: m*w**2*u
    F = lambda t: 0
    dt = 0.2
    T = 6
    #u, t = solver(I, V, m, b, s, F, dt, T, 'linear')
    u, t = solver_bwdamping(I, V, m, b, s, F, dt, T, 'linear')
    diff = np.abs(u_exact(t) - u).max()
    print diff
    tol = 1E-14
    #assert diff < tol

    #u, t = solver(I, V, m, b, s, F, dt, T, 'quadratic')
    u, t = solver_bwdamping(I, V, m, b, s, F, dt, T, 'quadratic')
    print diff
    diff = np.abs(u_exact(t) - u).max()
    assert diff < tol

def test_mms():
    """Use method of manufactured solutions."""
    m = 4.; b = 1
    w = 1.5
    t = sym.Symbol('t')
    u_exact = 3*sym.exp(-0.2*t)*sym.cos(1.2*t)
    I = u_exact.subs(t, 0).evalf()
    V = sym.diff(u_exact, t).subs(t, 0).evalf()
    u_exact_py = sym.lambdify(t, u_exact, modules='numpy')
    s = lambda u: u**3
    dt = 0.2
    T = 6
    errors_linear = []
    errors_quadratic = []
    # Run grid refinements and compute exact error
    for i in range(5):
        F_formula = lhs_eq(t, m, b, s, u_exact, 'linear')
        F = sym.lambdify(t, F_formula)
        #u1, t1 = solver(I, V, m, b, s, F, dt, T, 'linear')
        u1, t1 = solver_bwdamping(I, V, m, b, s,
                                  F, dt, T, 'linear')
        error = np.sqrt(np.sum((u_exact_py(t1) - u1)**2)*dt)
        errors_linear.append((dt, error))
```

```
        F_formula = lhs_eq(t, m, b, s, u_exact, 'quadratic')
        #print sym.latex(F_formula, mode='plain')
        F = sym.lambdify(t, F_formula)
        #u2, t2 = solver(I, V, m, b, s, F, dt, T, 'quadratic')
        u2, t2 = solver_bwdamping(I, V, m, b, s,
                                  F, dt, T, 'quadratic')
        error = np.sqrt(np.sum((u_exact_py(t2) - u2)**2)*dt)
        errors_quadratic.append((dt, error))
        dt /= 2
    # Estimate convergence rates
    tol = 0.05
    for errors in errors_linear, errors_quadratic:
        for i in range(1, len(errors)):
            dt, error = errors[i]
            dt_1, error_1 = errors[i-1]
            r = np.log(error/error_1)/np.log(dt/dt_1)
        # check r for final simulation with (final and) smallest dt
        # note that the method now is 1st order, i.e. r should
        # approach 1.0
        print r
        assert abs(r - 1.0) < tol

import os, sys
sys.path.insert(0, os.path.join(os.pardir, 'src-vib'))
from vib import (plot_empirical_freq_and_amplitude,
                 visualize_front, visualize_front_ascii,
                 minmax, periods, amplitudes,
                 solver as solver2)

def visualize(list_of_curves, legends, title='', filename='tmp'):
    """Plot list of curves: (u, t)."""
    for u, t in list_of_curves:
        plt.plot(t, u)
        plt.hold('on')
    plt.legend(legends)
    plt.xlabel('t')
    plt.ylabel('u')
    plt.title(title)
    plt.savefig(filename + '.png')
    plt.savefig(filename + '.pdf')
    plt.show()

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('--b', type=float, default=0.0)
    parser.add_argument('--s', type=str, default='4*pi**2*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=20)
    parser.add_argument('--window_width', type=float, default=30.,
                        help='Number of periods in a window')
    parser.add_argument('--damping', type=str, default='linear')
    parser.add_argument('--savefig', action='store_true')
    # Hack to allow --SCITOOLS options
    # (scitools.std reads this argument at import)
    parser.add_argument('--SCITOOLS_easyviz_backend',
                        default='matplotlib')
    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, b, dt, T, window_width, savefig, damping = \
       a.I, a.V, a.m, a.b, a.dt, a.T, a.window_width, a.savefig, \
       a.damping

    # compute u by both methods and then visualize the difference
    u, t = solver2(I, V, m, b, s, F, dt, T, damping)
    u_bw, _ = solver_bwdamping(I, V, m, b, s, F, dt, T, damping)
    u_diff = u - u_bw

    num_periods = plot_empirical_freq_and_amplitude(u_diff, t)
    if num_periods <= 40:
        plt.figure()
        legends = ['1st-2nd order method',
                   '2nd order method',
                   '1st order method']
        visualize([(u_diff, t), (u, t), (u_bw, t)], legends)
    else:
        visualize_front(u_diff, t, window_width, savefig)
        #visualize_front_ascii(u_diff, t)
    plt.show()

if __name__ == '__main__':
    main()
    #test_constant()
    #test_sinusoidal()
    #test_mms()
    #test_quadratic()
    raw_input()
```

Here is a comparison of standard method (2nd order) and backward differences for damping (1st order) for 10 (left) and 40 (right) time steps per period:



Filename: `vib_gen_bwdamping`.

Exercise 21: 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 10.10:

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

However, contrary to what is done in Section 10.10, we want to apply the idea of a forward-backward discretization: u is marched forward by a one-sided Forward Euler scheme applied to the first equation, and thereafter v can be marched forward by a Backward Euler scheme in the second equation. Express the idea in operator notation and write out the scheme. Unfortunately, the backward difference for the v equation creates a nonlinearity $|v^{n+1}|v^{n+1}$. To linearize this nonlinearity, use the known value v^n inside the absolute value factor, i.e., $|v^{n+1}|v^{n+1} \approx |v^n|v^{n+1}$. Show that the resulting scheme is equivalent to the one in Section 10.10 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 10.10) carefully apply centered differences and a geometric mean on a staggered mesh to achieve second-order accuracy. Filename: `vib_gen_bwdamping`.

12 Applications of vibration models

The following text derives some of the most well-known physical problems that lead to second-order ODE models of the type addressed in this document. We

consider a simple spring-mass system; thereafter extended with nonlinear spring, damping, and external excitation; a spring-mass system with sliding friction; a simple and a physical (classical) pendulum; and an elastic pendulum.

12.1 Oscillating mass attached to a spring

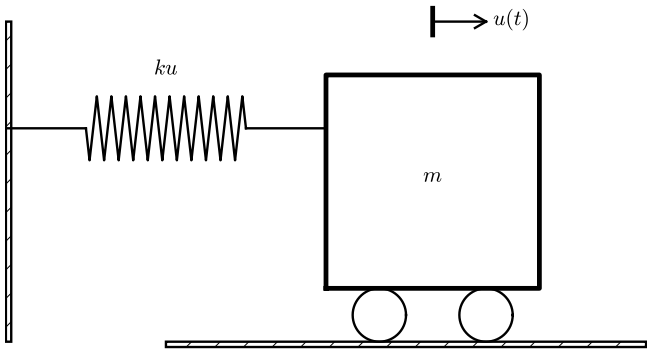


Figure 17: Simple oscillating mass.

The most fundamental mechanical vibration system is depicted in Figure 17. A body with mass m is attached to a spring and can move horizontally without friction (in the wheels). The position of the body is given by the vector $\boldsymbol{r}(t) = u(t)\boldsymbol{i}$, where $\boldsymbol{i}$ is a unit vector in x direction. There is only one force acting on the body: a spring force $\boldsymbol{F}_s = -ku\boldsymbol{i}$, where k is a constant. The point $x = 0$, where $u = 0$, must therefore correspond to the body's position where the spring is neither extended nor compressed, so the force vanishes.

The basic physical principle that governs the motion of the body is Newton's second law of motion: $\boldsymbol{F} = m\boldsymbol{a}$, where $\boldsymbol{F}$ is the sum of forces on the body, m is its mass, and $\boldsymbol{a} = \ddot{\boldsymbol{r}}$ is the acceleration. We use the dot for differentiation with respect to time, which is usual in mechanics. Newton's second law simplifies here to $-\boldsymbol{F}_s = m\ddot{u}\boldsymbol{i}$, which translates to

$$-ku = m\ddot{u}\,.$$

Two initial conditions are needed: $u(0) = I$, $\dot{u}(0) = V$. The ODE problem is normally written as

$$m\ddot{u} + ku = 0, \quad u(0) = I,\ \dot{u}(0) = V\,. \tag{123}$$

It is not uncommon to divide by m and introduce the frequency $\omega = \sqrt{k/m}$:

$$\ddot{u} + \omega^2 u = 0, \quad u(0) = I,\ \dot{u}(0) = V\,. \tag{124}$$

This is the model problem in the first part of this chapter, with the small difference that we write the time derivative of u with a dot above, while we used u' and u'' in previous parts of the document.

Since only one scalar mathematical quantity, $u(t)$, describes the complete motion, we say that the mechanical system has one degree of freedom (DOF).

Scaling. For numerical simulations it is very convenient to scale (124) and thereby get rid of the problem of finding relevant values for all the parameters m, k, I, and V. Since the amplitude of the oscillations are dictated by I and V (or more precisely, V/ω), we scale u by I (or V/ω if $I = 0$):

$$\bar{u} = \frac{u}{I}, \quad \bar{t} = \frac{t}{t_c}\,.$$

The time scale t_c is normally chosen as the inverse period $2\pi/\omega$ or angular frequency $1/\omega$, most often as $t_c = 1/\omega$. Inserting the dimensionless quantities $\bar{u}$ and $\bar{t}$ in (124) results in the scaled problem

$$\frac{d^2\bar{u}}{d\bar{t}^2} + \bar{u} = 0, \quad \bar{u}(0) = 1, \ \frac{\bar{u}}{\bar{t}}(0) = \beta = \frac{V}{I\omega},$$

where β is a dimensionless number. Any motion that starts from rest ($V = 0$) is free of parameters in the scaled model!

The physics. The typical physics of the system in Figure 17 can be described as follows. Initially, we displace the body to some position I, say at rest ($V = 0$). After releasing the body, the spring, which is extended, will act with a force $-kI\boldsymbol{i}$ and pull the body to the left. This force causes an acceleration and therefore increases velocity. The body passes the point $x = 0$, where $u = 0$, and the spring will then be compressed and act with a force $kx\boldsymbol{i}$ against the motion and cause retardation. At some point, the motion stops and the velocity is zero, before the spring force $kx\boldsymbol{i}$ has worked long enough to push the body in positive direction. The result is that the body accelerates back and forth. As long as there is no friction forces to damp the motion, the oscillations will continue forever.

12.2 General mechanical vibrating system

The mechanical system in Figure 17 can easily be extended to the more general system in Figure 18, where the body is attached to a spring and a dashpot, and also subject to an environmental force $F(t)\boldsymbol{i}$. The system has still only one degree of freedom since the body can only move back and forth parallel to the x axis. The spring force was linear, $\boldsymbol{F}_s = -ku\boldsymbol{i}$, in Section 12.1, but in more general cases it can depend nonlinearly on the position. We therefore set $\boldsymbol{F}_s = s(u)\boldsymbol{i}$. The dashpot, which acts as a damper, results in a force $\boldsymbol{F}_d$ that depends on the body's velocity $\dot{u}$ and that always acts against the motion. The mathematical model of the force is written $\boldsymbol{F}_d = f(\dot{u})\boldsymbol{i}$. A positive $\dot{u}$ must result in a force acting in the positive x direction. Finally, we have the external environmental force $\boldsymbol{F}_e = F(t)\boldsymbol{i}$.

Newton's second law of motion now involves three forces:

$$F(t)\boldsymbol{i} - f(\dot{u})\boldsymbol{i} - s(u)\boldsymbol{i} = m\ddot{u}\boldsymbol{i}\,.$$

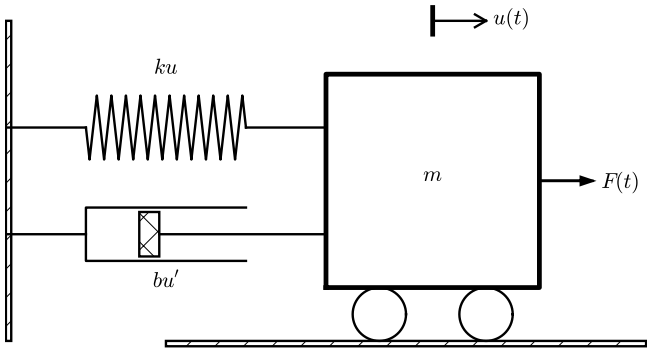


Figure 18: General oscillating system.

The common mathematical form of the ODE problem is

$$m\ddot{u} + f(\dot{u}) + s(u) = F(t), \quad u(0) = I,\ \dot{u}(0) = V\,. \tag{125}$$

This is the generalized problem treated in the last part of the present chapter, but with prime denoting the derivative instead of the dot.

The most common models for the spring and dashpot are linear: $f(\dot{u}) = b\dot{u}$ with a constant $b \geq 0$, and $s(u) = ku$ for a constant k.

Scaling. A specific scaling requires specific choices of f, s, and F. Suppose we have

$$f(\dot{u}) = b|\dot{u}|\dot{u}, \quad s(u) = ku, \quad F(t) = A\sin(\phi t)\,.$$

We introduce dimensionless variables as usual, $\bar{u} = u/u_c$ and $\bar{t} = t/t_c$. The scale u_c depends both on the initial conditions and F, but as time grows, the effect of the initial conditions die out and F will drive the motion. Inserting $\bar{u}$ and $\bar{t}$ in the ODE gives

$$m\frac{u_c}{t_c^2}\frac{d^2\bar{u}}{d\bar{t}^2} + b\frac{u_c^2}{t_c^2}\left|\frac{d\bar{u}}{d\bar{t}}\right|\frac{d\bar{u}}{d\bar{t}} + ku_c\bar{u} = A\sin(\phi t_c\bar{t})\,.$$

We divide by u_c/t_c^2 and demand the coefficients of the $\bar{u}$ and the forcing term from $F(t)$ to have unit coefficients. This leads to the scales

$$t_c = \sqrt{\frac{m}{k}}, \quad u_c = \frac{A}{k}\,.$$

The scaled ODE becomes

$$\frac{d^2\bar{u}}{d\bar{t}^2} + 2\beta\left|\frac{d\bar{u}}{d\bar{t}}\right|\frac{d\bar{u}}{d\bar{t}} + \bar{u} = \sin(\gamma\bar{t}), \tag{126}$$

where there are two dimensionless numbers:

$$\beta = \frac{Ab}{2mk}, \quad \gamma = \phi\sqrt{\frac{m}{k}}\,.$$

The β number measures the size of the damping term (relative to unity) and is assumed to be small, basically because b is small. The ϕ number is the ratio of the time scale of free vibrations and the time scale of the forcing. The scaled initial conditions have two other dimensionless numbers as values:

$$\bar{u}(0) = \frac{Ik}{A}, \quad \frac{d\bar{u}}{d\bar{t}} = \frac{t_c}{u_c}V = \frac{V}{A}\sqrt{mk}\,.$$

12.3 A sliding mass attached to a spring

Consider a variant of the oscillating body in Section 12.1 and Figure 17: the body rests on a flat surface, and there is sliding friction between the body and the surface. Figure 19 depicts the problem.

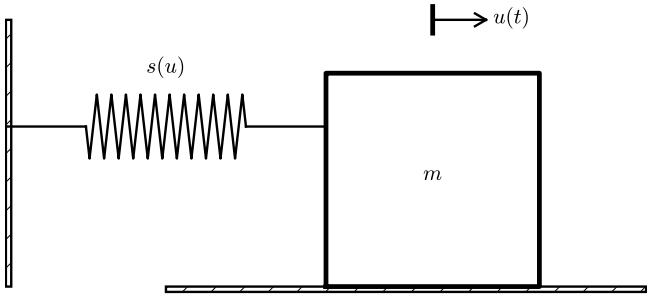


Figure 19: Sketch of a body sliding on a surface.

The body is attached to a spring with spring force $-s(u)\boldsymbol{i}$. The friction force is proportional to the normal force on the surface, $-mg\boldsymbol{j}$, and given by $-f(\dot{u})\boldsymbol{i}$, where

$$f(\dot{u}) = \begin{cases} -\mu mg, & \dot{u} < 0, \\ \mu mg, & \dot{u} > 0, \\ 0, & \dot{u} = 0 \end{cases}$$

Here, μ is a friction coefficient. With the signum function

$$\text{sign(x)} = \begin{cases} -1, & x < 0, \\ 1, & x > 0, \\ 0, & x = 0 \end{cases}$$

we can simply write $f(\dot{u}) = \mu mg\,\text{sign}(\dot{u})$ (the sign function is implemented by `numpy.sign`).

The equation of motion becomes

$$m\ddot{u} + \mu mg\text{sign}(\dot{u}) + s(u) = 0, \quad u(0) = I,\ \dot{u}(0) = V\,. \tag{127}$$

12.4 A jumping washing machine

A washing machine is placed on four springs with efficient dampers. If the machine contains just a few clothes, the circular motion of the machine induces a sinusoidal external force from the floor and the machine will jump up and down if the frequency of the external force is close to the natural frequency of the machine and its spring-damper system.

12.5 Motion of a pendulum

Simple pendulum. A classical problem in mechanics is the motion of a pendulum. We first consider a simplified pendulum (sometimes also called a mathematical pendulum): a small body of mass m is attached to a massless wire and can oscillate back and forth in the gravity field. Figure 20 shows a sketch of the problem.

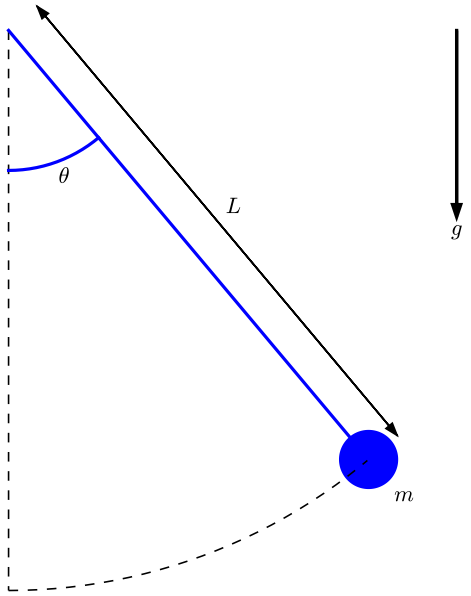


Figure 20: Sketch of a simple pendulum.

The motion is governed by Newton's 2nd law, so we need to find expressions for the forces and the acceleration. Three forces on the body are considered: an unknown force S from the wire, the gravity force mg, and an air resistance force, $\frac{1}{2}C_D\varrho A|v|v$, hereafter called the drag force, directed against the velocity of the body. Here, C_D is a drag coefficient, ϱ is the density of air, A is the cross section area of the body, and v is the magnitude of the velocity.

We introduce a coordinate system with polar coordinates and unit vectors $\boldsymbol{i}_r$ and $\boldsymbol{i}_\theta$ as shown in Figure 21. The position of the center of mass of the body is

$$\boldsymbol{r}(t) = x_0\boldsymbol{i} + y_0\boldsymbol{j} + L\boldsymbol{i}_r,$$

where $\boldsymbol{i}$ and $\boldsymbol{j}$ are unit vectors in the corresponding Cartesian coordinate system in the x and y directions, respectively. We have that $\boldsymbol{i}_r = \cos\theta\boldsymbol{i} + \sin\theta\boldsymbol{j}$.

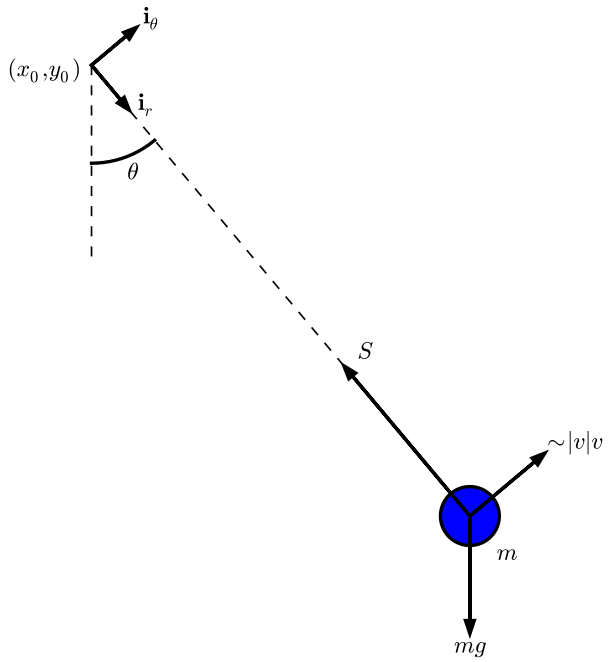


Figure 21: Forces acting on a simple pendulum.

The forces are now expressed as follows.

- Wire force: $-S\boldsymbol{i}_r$
- Gravity force: $-mg\boldsymbol{j} = mg(-\sin\theta\,\boldsymbol{i}_\theta + \cos\theta\,\boldsymbol{i}_r)$
- Drag force: $-\frac{1}{2}C_D\varrho A|v|v\,\boldsymbol{i}_\theta$

Since a positive velocity means movement in the direction of $\boldsymbol{i}_\theta$, the drag force must be directed along $-\boldsymbol{i}_\theta$ so it works against the motion. We assume motion in air so that the added mass effect can be neglected (for a spherical body, the added mass is $\frac{1}{2}\varrho V$, where V is the volume of the body). Also the buoyancy effect can be neglected for motion in the air when the density difference between the fluid and the body is so significant.

The velocity of the body is found from $\boldsymbol{r}$:

$$\boldsymbol{v}(t) = \dot{\boldsymbol{r}}(t) = \frac{d}{d\theta}(x_0\boldsymbol{i} + y_0\boldsymbol{j} + L\boldsymbol{i}_r)\frac{d\theta}{dt} = L\dot{\theta}\boldsymbol{i}_\theta,$$

since $\frac{d}{d\theta}\boldsymbol{i}_r = \boldsymbol{i}_\theta$. It follows that $v = |\boldsymbol{v}| = L\dot{\theta}$. The acceleration is

$$\boldsymbol{a}(t) = \dot{\boldsymbol{v}}(r) = \frac{d}{dt}(L\dot{\theta}\boldsymbol{i}_\theta) = L\ddot{\theta}\boldsymbol{i}_\theta + L\dot{\theta}\frac{d\boldsymbol{i}_\theta}{d\theta}\dot{\theta} == L\ddot{\theta}\boldsymbol{i}_\theta - L\dot{\theta}^2\boldsymbol{i}_r,$$

since $\frac{d}{d\theta}\boldsymbol{i}_\theta = -\boldsymbol{i}_r$.

Newton's 2nd law of motion becomes

$$-S\boldsymbol{i}_r + mg(-\sin\theta\,\boldsymbol{i}_\theta + \cos\theta\,\boldsymbol{i}_r) - \frac{1}{2}C_D\varrho AL^2|\dot{\theta}|\dot{\theta}\,\boldsymbol{i}_\theta = mL\ddot{\theta}\dot{\theta}\,\boldsymbol{i}_\theta - L\dot{\theta}^2\boldsymbol{i}_r,$$

leading to two component equations

$$-S + mg\cos\theta = -L\dot{\theta}^2, \tag{128}$$

$$-mg\sin\theta - \frac{1}{2}C_D\varrho AL^2|\dot{\theta}|\dot{\theta} = mL\ddot{\theta}\,. \tag{129}$$

From (128) we get an expression for $S = mg\cos\theta + L\dot{\theta}^2$, and from (129) we get a differential equation for the angle $\theta(t)$. This latter equation is ordered as

$$m\ddot{\theta} + \frac{1}{2}C_D\varrho AL|\dot{\theta}|\dot{\theta} + \frac{mg}{L}\sin\theta = 0\,. \tag{130}$$

Two initial conditions are needed: $\theta = \Theta$ and $\dot{\theta} = \Omega$. Normally, the pendulum motion is started from rest, which means $\Omega = 0$.

Equation (130) fits the general model used in (75) in Section 10 if we define $u = \theta$, $f(u') = \frac{1}{2}C_D\varrho AL|\dot{u}|\dot{u}$, $s(u) = L^{-1}mg\sin u$, and $F = 0$. If the body is a sphere with radius R, we can take $C_D = 0.4$ and $A = \pi R^2$. Exercise 25 asks you to scale the equations and carry out specific simulations with this model.

Physical pendulum. The motion of a compound or physical pendulum where the wire is a rod with mass, can be modeled very similarly. The governing equation is $I\boldsymbol{a} = \boldsymbol{T}$ where I is the moment of inertia of the entire body about the point (x_0, y_0), and $\boldsymbol{T}$ is the sum of moments of the forces with respect to (x_0, y_0). The vector equation reads

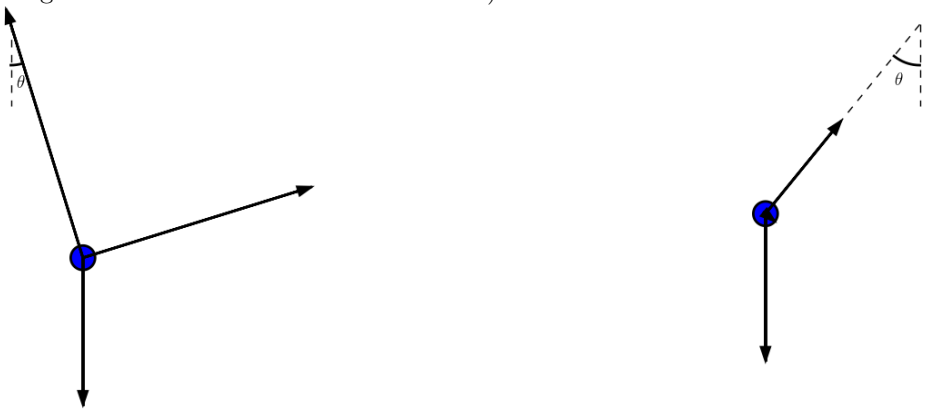
$$\boldsymbol{r}\times(-S\boldsymbol{i}_r + mg(-\sin\theta\boldsymbol{i}_\theta + \cos\theta\boldsymbol{i}_r) - \frac{1}{2}C_D\varrho AL^2|\dot{\theta}|\dot{\theta}\boldsymbol{i}_\theta) = I(L\ddot{\theta}\dot{\theta}\boldsymbol{i}_\theta - L\dot{\theta}^2\boldsymbol{i}_r)\,.$$

The component equation in $\boldsymbol{i}_\theta$ direction gives the equation of motion for $\theta(t)$:

$$I\ddot{\theta} + \frac{1}{2}C_D\varrho AL^3|\dot{\theta}|\dot{\theta} + mgL\sin\theta = 0\,. \tag{131}$$

12.6 Dynamic free body diagram during pendulum motion

Usually one plots the mathematical quantities as functions of time to visualize the solution of ODE models. Exercise 25 asks you to do this for the motion of a pendulum in the previous section. However, sometimes it is more instructive to look at other types of visualizations. For example, we have the pendulum and the free body diagram in Figures 20 and 21. We may think of these figures as animations in time instead. Especially the free body diagram will show both the motion of the pendulum *and* the size of the forces during the motion. The present section exemplifies how to make such a dynamic body diagram. Two typical snapshots of free body diagrams are displayed below (the drag force is magnified 5 times to become more visual!).



Dynamic physical sketches, coupled to the numerical solution of differential equations, requires a program to produce a sketch for the situation at each time level. Pysketcher is such a tool. In fact (and not surprising!) Figures 20 and 21 were drawn using Pysketcher. The details of the drawings are explained in the Pysketcher tutorial. Here, we outline how this type of sketch can be used to create an animated free body diagram during the motion of a pendulum.

Pysketcher is actually a layer of useful abstractions on top of standard plotting packages. This means that we in fact apply Matplotlib to make the animated free body diagram, but instead of dealing with a wealth of detailed Matplotlib commands, we can express the drawing in terms of more high-level objects, e.g., objects for the wire, angle θ, body with mass m, arrows for forces, etc. When the position of these objects are given through variables, we can just couple those variables to the dynamic solution of our ODE and thereby make a unique drawing for each θ value in a simulation.

Writing the solver. Let us start with the most familiar part of the current problem: writing the solver function. We use Odespy for this purpose. We also work with dimensionless equations. Since θ can be viewed as dimensionless, we only need to introduce a dimensionless time, here taken as $\bar{t} = t/\sqrt{L/g}$. The resulting dimensionless mathematical model for θ, the dimensionless angular

velocity ω, the dimensionless wire force $\bar{S}$, and the dimensionless drag force $\bar{D}$ is then

$$\frac{d\omega}{d\bar{t}} = -\alpha|\omega|\omega - \sin\theta, \tag{132}$$

$$\frac{d\theta}{d\bar{t}} = \omega, \tag{133}$$

$$\bar{S} = \omega^2 + \cos\theta, \tag{134}$$

$$\bar{D} = -\alpha|\omega|\omega, \tag{135}$$

with

$$\alpha = \frac{C_D \varrho \pi R^2 L}{2m}\,.$$

as a dimensionless parameter expressing the ratio of the drag force and the gravity force. The dimensionless ω is made non-dimensional by the time, so $\omega\sqrt{L/g}$ is the corresponding angular frequency with dimensions.

A suitable function for computing (132)-(135) is listed below.

```
def simulate(alpha, Theta, dt, T):
    import odespy

    def f(u, t, alpha):
        omega, theta = u
        return [-alpha*omega*abs(omega) - sin(theta),
                omega]

    import numpy as np
    Nt = int(round(T/float(dt)))
    t = np.linspace(0, Nt*dt, Nt+1)
    solver = odespy.RK4(f, f_args=[alpha])
    solver.set_initial_condition([0, Theta])
    u, t = solver.solve(
        t, terminate=lambda u, t, n: abs(u[n,1]) < 1E-3)
    omega = u[:,0]
    theta = u[:,1]
    S = omega**2 + np.cos(theta)
    drag = -alpha*np.abs(omega)*omega
    return t, theta, omega, S, drag
```

Drawing the free body diagram. The `sketch` function below applies Pysketcher objects to build a diagram like that in Figure 21, except that we have removed the rotation point (x_0, y_0) and the unit vectors in polar coordinates as these objects are not important for an animated free body diagram.

```
import sys
try:
```

```
    from pysketcher import *
except ImportError:
    print 'Pysketcher must be installed from'
    print 'https://github.com/hplgit/pysketcher'
    sys.exit(1)

# Overall dimensions of sketch
H = 15.
W = 17.

drawing_tool.set_coordinate_system(
    xmin=0, xmax=W, ymin=0, ymax=H,
    axis=False)

def sketch(theta, S, mg, drag, t, time_level):
    """
    Draw pendulum sketch with body forces at a time level
    corresponding to time t. The drag force is in
    drag[time_level], the force in the wire is S[time_level],
    the angle is theta[time_level].
    """
    import math
    a = math.degrees(theta[time_level])  # angle in degrees
    L = 0.4*H         # Length of pendulum
    P = (W/2, 0.8*H)  # Fixed rotation point

    mass_pt = path.geometric_features()['end']
    rod = Line(P, mass_pt)

    mass = Circle(center=mass_pt, radius=L/20.)
    mass.set_filled_curves(color='blue')
    rod_vec = rod.geometric_features()['end'] - \
              rod.geometric_features()['start']
    unit_rod_vec = unit_vec(rod_vec)
    mass_symbol = Text('$m$', mass_pt + L/10*unit_rod_vec)

    rod_start = rod.geometric_features()['start']  # Point P
    vertical = Line(rod_start, rod_start + point(0,-L/3))

    def set_dashed_thin_blackline(*objects):
        """Set linestyle of objects to dashed, black, width=1."""
        for obj in objects:
            obj.set_linestyle('dashed')
            obj.set_linecolor('black')
            obj.set_linewidth(1)

    set_dashed_thin_blackline(vertical)
    set_dashed_thin_blackline(rod)
    angle = Arc_wText(r'$\theta$', rod_start, L/6, -90, a,
                      text_spacing=1/30.)
```

```
    magnitude = 1.2*L/2   # length of a unit force in figure
    force = mg[time_level]  # constant (scaled eq: about 1)
    force *= magnitude
    mg_force  = Force(mass_pt, mass_pt + force*point(0,-1),
                      '', text_pos='end')
    force = S[time_level]
    force *= magnitude
    rod_force = Force(mass_pt, mass_pt - force*unit_vec(rod_vec),
                      '', text_pos='end',
                      text_spacing=(0.03, 0.01))
    force = drag[time_level]
    force *= magnitude
    air_force = Force(mass_pt, mass_pt -
                      force*unit_vec((rod_vec[1], -rod_vec[0])),
                      '', text_pos='end',
                      text_spacing=(0.04,0.005))

    body_diagram = Composition(
        {'mg': mg_force, 'S': rod_force, 'air': air_force,
         'rod': rod, 'body': mass
         'vertical': vertical, 'theta': angle,})

    body_diagram.draw(verbose=0)
    drawing_tool.savefig('tmp_%04d.png' % time_level, crop=False)
    # (No cropping: otherwise movies will be very strange!)
```

Making the animated free body diagram. It now remains to couple the `simulate` and `sketch` functions. We first run `simulate`:

```
from math import pi, radians, degrees
import numpy as np
alpha = 0.4
period = 2*pi   # Use small theta approximation
T = 12*period   # Simulate for 12 periods
dt = period/40  # 40 time steps per period
a = 70          # Initial amplitude in degrees
Theta = radians(a)

t, theta, omega, S, drag = simulate(alpha, Theta, dt, T)
```

The next step is to run through the time levels in the simulation and make a sketch at each level:

```
for time_level, t_ in enumerate(t):
    sketch(theta, S, mg, drag, t_, time_level)
```

The individual sketches are (by the `sketch` function) saved in files with names `tmp_%04d.png`. These can be combined to videos using (e.g.) `ffmpeg`. A

complete function `animate` for running the simulation and creating video files is listed below.

```
def animate():
    # Clean up old plot files
    import os, glob
    for filename in glob.glob('tmp_*.png') + glob.glob('movie.*'):
        os.remove(filename)
    # Solve problem
    from math import pi, radians, degrees
    import numpy as np
    alpha = 0.4
    period = 2*pi   # Use small theta approximation
    T = 12*period   # Simulate for 12 periods
    dt = period/40  # 40 time steps per period
    a = 70          # Initial amplitude in degrees
    Theta = radians(a)

    t, theta, omega, S, drag = simulate(alpha, Theta, dt, T)

    # Visualize drag force 5 times as large
    drag *= 5
    mg = np.ones(S.size)  # Gravity force (needed in sketch)

    # Draw animation
    import time
    for time_level, t_ in enumerate(t):
        sketch(theta, S, mg, drag, t_, time_level)
        time.sleep(0.2)  # Pause between each frame on the screen

    # Make videos
    prog = 'ffmpeg'
    filename = 'tmp_%04d.png'
    fps = 6
    codecs = {'flv': 'flv', 'mp4': 'libx264',
              'webm': 'libvpx', 'ogg': 'libtheora'}
    for ext in codecs:
        lib = codecs[ext]
        cmd = '%(prog)s -i %(filename)s -r %(fps)s ' % vars()
        cmd += '-vcodec %(lib)s movie.%(ext)s' % vars()
        print(cmd)
        os.system(cmd)
```

12.7 Motion of an elastic pendulum

Consider a pendulum as in Figure 20, but this time the wire is elastic. The length of the wire when it is not stretched is L_0, while $L(t)$ is the stretched length at time t during the motion.

Stretching the elastic wire a distance ΔL gives rise to a spring force $k\Delta L$ in the opposite direction of the stretching. Let $\boldsymbol{n}$ be a unit normal vector along the wire from the point $\boldsymbol{r}_0 = (x_0, y_0)$ and in the direction of $\boldsymbol{i}_\theta$, see Figure 21 for definition of (x_0, y_0) and $\boldsymbol{i}_\theta$. Obviously, we have $\boldsymbol{n} = \boldsymbol{i}_\theta$, but in this modeling of an elastic pendulum we do not need polar coordinates. Instead, it is more straightforward to develop the equation in Cartesian coordinates.

A mathematical expression for $\boldsymbol{n}$ is

$$\boldsymbol{n} = \frac{\boldsymbol{r} - \boldsymbol{r}_0}{L(t)},$$

where $L(t) = ||\boldsymbol{r} - \boldsymbol{r}_0||$ is the current length of the elastic wire. The position vector $\boldsymbol{r}$ in Cartesian coordinates reads $\boldsymbol{r}(t) = x(t)\boldsymbol{i} + y(t)\boldsymbol{j}$, where $\boldsymbol{i}$ and $\boldsymbol{j}$ are unit vectors in the x and y directions, respectively. It is convenient to introduce the Cartesian components n_x and n_y of the normal vector:

$$\boldsymbol{n} = \frac{\boldsymbol{r} - \boldsymbol{r}_0}{L(t)} = \frac{x(t) - x_0}{L(t)}\boldsymbol{i} + \frac{y(t) - y_0}{L(t)}\boldsymbol{j} = n_x\boldsymbol{i} + n_y\boldsymbol{j}\,.$$

The stretch ΔL in the wire is

$$\Delta t = L(t) - L_0\,.$$

The force in the wire is then $-S\boldsymbol{n} = -k\Delta L\boldsymbol{n}$.

The other forces are the gravity and the air resistance, just as in Figure 21. For motion in air we can neglect the added mass and buoyancy effects. The main difference is that we have a *model* for S in terms of the motion (as soon as we have expressed ΔL by $\boldsymbol{r}$). For simplicity, we drop the air resistance term (but Exercise 27 asks you to include it).

Newton's second law of motion applied to the body now results in

$$m\ddot{\boldsymbol{r}} = -k(L - L_0)\boldsymbol{n} - mg\boldsymbol{j} \tag{136}$$

The two components of (136) are

$$\ddot{x} = -\frac{k}{m}(L - L_0)n_x, \tag{137}$$

$$\tag{138}$$

$$\ddot{y} = -\frac{k}{m}(L - L_0)n_y - g\,. \tag{139}$$

Remarks about an elastic vs a non-elastic pendulum. Note that the derivation of the ODEs for an elastic pendulum is more straightforward than for a classical, non-elastic pendulum, since we avoid the details with polar coordinates, but instead work with Newton's second law directly in Cartesian coordinates. The reason why we can do this is that the elastic pendulum undergoes a general two-dimensional motion where all the forces are known or expressed as functions of $x(t)$ and $y(t)$, such that we get two ordinary differential equations. The motion

of the non-elastic pendulum, on the other hand, is constrained: the body has to move along a circular path, and the force S in the wire is unknown.

The non-elastic pendulum therefore leads to a *differential-algebraic* equation, i.e., ODEs for $x(t)$ and $y(t)$ combined with an extra constraint $(x-x_0)^2 + (y-y_0)^2 = L^2$ ensuring that the motion takes place along a circular path. The extra constraint (equation) is compensated by an extra unknown force $-S\boldsymbol{n}$. Differential-algebraic equations are normally hard to solve, especially with pen and paper. Fortunately, for the non-elastic pendulum we can do a trick: in polar coordinates the unknown force S appears only in the radial component of Newton's second law, while the unknown degree of freedom for describing the motion, the angle $\theta(t)$, is completely governed by the asimuthal component. This allows us to decouple the unknowns S and θ. But this is a kind of trick and not a widely applicable method. With an elastic pendulum we use straightforward reasoning with Newton's 2nd law and arrive at a standard ODE problem that (after scaling) is easy solve on a computer.

Initial conditions. What is the initial position of the body? We imagine that first the pendulum hangs in equilibrium in its vertical position, and then it is displaced an angle Θ. The equilibrium position is governed by the ODEs with the accelerations set to zero. The x component leads to $x(t) = x_0$, while the y component gives

$$0 = -\frac{k}{m}(L - L_0)n_y - g = \frac{k}{m}(L(0) - L_0) - g \quad \Rightarrow \quad L(0) = L_0 + mg/k,$$

since $n_y = -11$ in this position. The corresponding y value is then from $n_y = -1$:

$$y(t) = y_0 - L(0) = y_0 - (L_0 + mg/k)\,.$$

Let us now choose (x_0, y_0) such that the body is at the origin in the equilibrium position:

$$x_0 = 0, \quad y_0 = L_0 + mg/k\,.$$

Displacing the body an angle Θ to the right leads to the initial position

$$x(0) = (L_0 + mg/k)\sin\Theta, \quad y(0) = (L_0 + mg/k)(1 - \cos\Theta)\,.$$

The initial velocities can be set to zero: $x'(0) = y'(0) = 0$.

The complete ODE problem. We can summarize all the equations as follows:

$$
\begin{aligned}
\ddot{x} &= -\frac{k}{m}(L - L_0)n_x, \\
\ddot{y} &= -\frac{k}{m}(L - L_0)n_y - g, \\
L &= \sqrt{(x - x_0)^2 + (y - y_0)^2}, \\
n_x &= \frac{x - x_0}{L}, \\
n_y &= \frac{y - y_0}{L}, \\
x(0) &= (L_0 + mg/k)\sin\Theta, \\
x'(0) &= 0, \\
y(0) &= (L_0 + mg/k)(1 - \cos\Theta), \\
y'(0) &= 0\,.
\end{aligned}
$$

We insert n_x and n_y in the ODEs:

$$
\ddot{x} = -\frac{k}{m}\left(1 - \frac{L_0}{L}\right)(x - x_0), \tag{140}
$$

$$
\ddot{y} = -\frac{k}{m}\left(1 - \frac{L_0}{L}\right)(y - y_0) - g, \tag{141}
$$

$$
L = \sqrt{(x - x_0)^2 + (y - y_0)^2}, \tag{142}
$$

$$
x(0) = (L_0 + mg/k)\sin\Theta, \tag{143}
$$

$$
x'(0) = 0, \tag{144}
$$

$$
y(0) = (L_0 + mg/k)(1 - \cos\Theta), \tag{145}
$$

$$
y'(0) = 0\,. \tag{146}
$$

Scaling. The elastic pendulum model can be used to study both an elastic pendulum and a classic, non-elastic pendulum. The latter problem is obtained by letting $k \to \infty$. Unfortunately, a serious problem with the ODEs (140)-(141) is that for large k, we have a very large factor k/m multiplied by a very small number $1 - L_0/L$, since for large k, $L \approx L_0$ (very small deformations of the wire). The product is subject to significant round-off errors for many relevant physical values of the parameters. To circumvent the problem, we introduce a scaling. This will also remove physical parameters from the problem such that we end up with only one dimensionless parameter, closely related to the elasticity of the wire. Simulations can then be done by setting just this dimensionless parameter.

The characteristic length can be taken such that in equilibrium, the scaled length is unity, i.e., the characteristic length is $L_0 + mg/k$:

$$
\bar{x} = \frac{x}{L_0 + mg/k}, \quad \bar{y} = \frac{y}{L_0 + mg/k}\,.
$$

We must then also work with the scaled length $\bar{L} = L/(L_0 + mg/k)$.

Introducing $\bar{t} = t/t_c$, where t_c is a characteristic time we have to decide upon later, one gets

$$\begin{aligned}
\frac{d^2\bar{x}}{d\bar{t}^2} &= -t_c^2 \frac{k}{m}\left(1 - \frac{L_0}{L_0 + mg/k}\frac{1}{\bar{L}}\right)\bar{x},\\
\frac{d^2\bar{y}}{d\bar{t}^2} &= -t_c^2 \frac{k}{m}\left(1 - \frac{L_0}{L_0 + mg/k}\frac{1}{\bar{L}}\right)(\bar{y} - 1) - t_c^2 \frac{g}{L_0 + mg/k},\\
\bar{L} &= \sqrt{\bar{x}^2 + (\bar{y} - 1)^2},\\
\bar{x}(0) &= \sin\Theta,\\
\bar{x}'(0) &= 0,\\
\bar{y}(0) &= 1 - \cos\Theta,\\
\bar{y}'(0) &= 0\,.
\end{aligned}$$

For a non-elastic pendulum with small angles, we know that the frequency of the oscillations are $\omega = \sqrt{L/g}$. It is therefore natural to choose a similar expression here, either the length in the equilibrium position,

$$t_c^2 = \frac{L_0 + mg/k}{g}\,.$$

or simply the unstretched length,

$$t_c^2 = \frac{L_0}{g}\,.$$

These quantities are not very different (since the elastic model is valid only for quite small elongations), so we take the latter as it is the simplest one.

The ODEs become

$$\begin{aligned}
\frac{d^2\bar{x}}{d\bar{t}^2} &= -\frac{L_0 k}{mg}\left(1 - \frac{L_0}{L_0 + mg/k}\frac{1}{\bar{L}}\right)\bar{x},\\
\frac{d^2\bar{y}}{d\bar{t}^2} &= -\frac{L_0 k}{mg}\left(1 - \frac{L_0}{L_0 + mg/k}\frac{1}{\bar{L}}\right)(\bar{y} - 1) - \frac{L_0}{L_0 + mg/k},\\
\bar{L} &= \sqrt{\bar{x}^2 + (\bar{y} - 1)^2}\,.
\end{aligned}$$

We can now identify a dimensionless number

$$\beta = \frac{L_0}{L_0 + mg/k} = \frac{1}{1 + \frac{mg}{L_0 k}},$$

which is the ratio of the unstretched length and the stretched length in equilibrium. The non-elastic pendulum will have $\beta = 1$ ($k \to \infty$). With β the ODEs read

$$\frac{d^2\bar{x}}{d\bar{t}^2} = -\frac{\beta}{1-\beta}\left(1 - \frac{\beta}{\bar{L}}\right)\bar{x}, \tag{147}$$

$$\frac{d^2\bar{y}}{d\bar{t}^2} = -\frac{\beta}{1-\beta}\left(1 - \frac{\beta}{\bar{L}}\right)(\bar{y}-1) - \beta, \tag{148}$$

$$\bar{L} = \sqrt{\bar{x}^2 + (\bar{y}-1)^2}, \tag{149}$$

$$\bar{x}(0) = (1+\epsilon)\sin\Theta, \tag{150}$$

$$\frac{d\bar{x}}{d\bar{t}}(0) = 0, \tag{151}$$

$$\bar{y}(0) = 1 - (1+\epsilon)\cos\Theta, \tag{152}$$

$$\frac{d\bar{y}}{d\bar{t}}(0) = 0, \tag{153}$$

We have here added a parameter ϵ, which is an additional downward stretch of the wire at $t = 0$. This parameter makes it possible to do a desired test: vertical oscillations of the pendulum. Without ϵ, starting the motion from $(0, 0)$ with zero velocity will result in $x = y = 0$ for all times (also a good test!), but with an initial stretch so the body's position is $(0, \epsilon)$, we will have oscillatory vertical motion with amplitude ϵ (see Exercise 26).

Remark on the non-elastic limit. We immediately see that as $k \to \infty$ (i.e., we obtain a non-elastic pendulum), $\beta \to 1$, $\bar{L} \to 1$, and we have very small values $1 - \beta\bar{L}^{-1}$ divided by very small values $1 - \beta$ in the ODEs. However, it turns out that we can set β very close to one and obtain a path of the body that within the visual accuracy of a plot does not show any elastic oscillations. (Should the division of very small values become a problem, one can study the limit by L'Hospital's rule:

$$\lim_{\beta\to 1} \frac{1 - \beta\bar{L}^{-1}}{1-\beta} = \frac{1}{\bar{L}},$$

and use the limit $\bar{L}^{-1}$ in the ODEs for β values very close to 1.)

12.8 Vehicle on a bumpy road

We consider a very simplistic vehicle, on one wheel, rolling along a bumpy road. The oscillatory nature of the road will induce an external forcing on the spring system in the vehicle and cause vibrations. Figure 22 outlines the situation.

To derive the equation that governs the motion, we must first establish the position vector of the black mass at the top of the spring. Suppose the spring has length L without any elongation or compression, suppose the radius of the wheel is R, and suppose the height of the black mass at the top is H. With the aid of the $\boldsymbol{r}_0$ vector in Figure 22, the position $\boldsymbol{r}$ of the center point of the mass is

$$\boldsymbol{r} = \boldsymbol{r}_0 + 2R\boldsymbol{j} + L\boldsymbol{j} + u\boldsymbol{j} + \frac{1}{2}H\boldsymbol{j}, \tag{154}$$

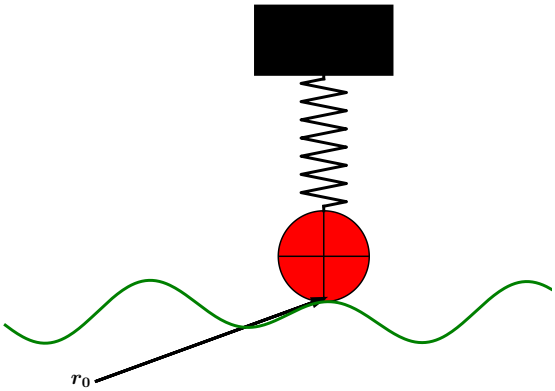


Figure 22: Sketch of one-wheel vehicle on a bumpy road.

where u is the elongation or compression in the spring according to the (unknown and to be computed) vertical displacement u relative to the road. If the vehicle travels with constant horizontal velocity v and $h(x)$ is the shape of the road, then the vector $\boldsymbol{r}_0$ is

$$\boldsymbol{r}_0 = vt\boldsymbol{i} + h(vt)\boldsymbol{j},$$

if the motion starts from $x = 0$ at time $t = 0$.

The forces on the mass is the gravity, the spring force, and an optional damping force that is proportional to the vertical velocity $\dot{u}$. Newton's second law of motion then tells that

$$m\ddot{\boldsymbol{r}} = -mg\boldsymbol{j} - s(u) - b\dot{u}\boldsymbol{j}\,.$$

This leads to

$$m\ddot{u} = -s(u) - b\dot{u} - mg - mh''(vt)v^2$$

To simplify a little bit, we omit the gravity force mg in comparison with the other terms. Introducing u' for $\dot{u}$ then gives a standard damped, vibration equation with external forcing:

$$mu'' + bu' + s(u) = -mh''(vt)v^2\,. \tag{155}$$

Since the road is normally known just as a set of array values, h'' must be computed by finite differences. Let Δx be the spacing between measured values $h_i = h(i\Delta x)$ on the road. The discrete second-order derivative h'' reads

$$q_i = \frac{h_{i-1} - 2h_i + h_{i+1}}{\Delta x^2}, \quad i = 1, \ldots, N_x - 1\,.$$

We may for maximum simplicity set the end points as $q_0 = q_1$ and $q_{N_x} = q_{N_x-1}$. The term $-mh''(vt)v^2$ corresponds to a force with discrete time values

$$F^n = -mq_n v^2, \quad \Delta t = v^{-1}\Delta x\,.$$

This force can be directly used in a numerical model

$$[mD_tD_tu + bD_{2t}u + s(u) = F]^n\,.$$

Software for computing u and also making an animated sketch of the motion like we did in Section 12.6 is found in a separate project on the web: `https://github.com/hplgit/bumpy`. You may start looking at the tutorial.

12.9 Bouncing ball

A bouncing ball is a ball in free vertically fall until it impacts the ground, but during the impact, some kinetic energy is lost, and a new motion upwards with reduced velocity starts. After the motion is retarded, a new free fall starts, and the process is repeated. At some point the velocity close to the ground is so small that the ball is considered to be finally at rest.

The motion of the ball falling in air is governed by Newton's second law $F = ma$, where a is the acceleration of the body, m is the mass, and F is the sum of all forces. Here, we neglect the air resistance so that gravity $-mg$ is the only force. The height of the ball is denoted by h and v is the velocity. The relations between h, v, and a,

$$h'(t) = v(t), \quad v'(t) = a(t),$$

combined with Newton's second law gives the ODE model

$$h''(t) = -g, \tag{156}$$

or expressed alternatively as a system of first-order equations:

$$v'(t) = -g, \tag{157}$$
$$h'(t) = v(t)\,. \tag{158}$$

These equations govern the motion as long as the ball is away from the ground by a small distance $\epsilon_h > 0$. When $h < \epsilon_h$, we have two cases.

1. The ball impacts the ground, recognized by a sufficiently large negative velocity ($v < -\epsilon_v$). The velocity then changes sign and is reduced by a factor C_R, known as the coefficient of restitution. For plotting purposes, one may set $h = 0$.

2. The motion stops, recognized by a sufficiently small velocity ($|v| < \epsilon_v$) close to the ground.

12.10 Two-body gravitational problem

Consider two astronomical objects A and B that attract each other by gravitational forces. A and B could be two stars in a binary system, a planet orbiting a star, or a moon orbiting a planet. Each object is acted upon by the gravitational force due to the other object. Consider motion in a plane (for simplicity) and let (x_A, y_A) and (x_B, y_B) be the positions of object A and B, respectively.

The governing equations. Newton's second law of motion applied to each object is all we need to set up a mathematical model for this physical problem:

$$m_A \ddot{\boldsymbol{x}}_A = \boldsymbol{F}, \tag{159}$$
$$m_B \ddot{\boldsymbol{x}}_B = -\boldsymbol{F}, \tag{160}$$

where F is the gravitational force

$$\boldsymbol{F} = \frac{G m_A m_B}{||\boldsymbol{r}||^3}\boldsymbol{r},$$

where

$$\boldsymbol{r}(t) = \boldsymbol{x}_B(t) - \boldsymbol{x}_A(t),$$

and G is the gravitational constant: $G = 6.674 \cdot 10^{-11}$ Nm^2/kg^2.

Scaling. A problem with these equations is that the parameters are very large (m_A, m_B, $||\boldsymbol{r}||$) or very small (G). The rotation time for binary stars can be very small and large as well. It is therefore advantageous to scale the equations. A natural length scale could be the initial distance between the objects: $L = \boldsymbol{r}(0)$. We write the dimensionless quantities as

$$\bar{\boldsymbol{x}}_A = \frac{\boldsymbol{x}_A}{L}, \quad \bar{\boldsymbol{x}}_B = \frac{\boldsymbol{x}_B}{L}, \quad \bar{t} = \frac{t}{t_c}\,.$$

The gravity force is transformed to

$$\boldsymbol{F} = \frac{G m_A m_B}{L^2 ||\bar{\boldsymbol{r}}||^3}\bar{\boldsymbol{r}}, \quad \bar{\boldsymbol{r}} = \bar{\boldsymbol{x}}_B - \bar{\boldsymbol{x}}_A,$$

so the first ODE for $\boldsymbol{x}_A$ becomes

$$\frac{d^2\bar{\boldsymbol{x}}_A}{d\bar{t}^2} = \frac{G m_B t_c^2}{L^3}\frac{\bar{\boldsymbol{r}}}{||\bar{\boldsymbol{r}}||^3}\,.$$

Assuming that quantities with a bar and their derivatives are around unity in size, it is natural to choose t_c such that the fraction $G m_B t_c / L^2 = 1$:

$$t_c = \sqrt{\frac{L^3}{G m_B}}\,.$$

From the other equation for $\boldsymbol{x}_B$ we get another candidate for t_c with m_A instead of m_B. Which mass we choose play a role if $m_A \ll m_B$ or $m_B \ll m_A$. One solution is to use the sum of the masses:

$$t_c = \sqrt{\frac{L^3}{G(m_A + m_B)}}\,.$$

Taking a look at Kepler's laws of planetary motion, the orbital period for a planet around the star is given by the t_c above, except for a missing factor of 2π, but that means that t_c^{-1} is just the angular frequency of the motion. Our characteristic time t_c is therefore highly relevant. Introducing the dimensionless number

$$\alpha = \frac{m_A}{m_B},$$

we can write the dimensionless ODE as

$$\frac{d^2\bar{\boldsymbol{x}}_A}{d\bar{t}^2} = \frac{1}{1+\alpha}\frac{\bar{\boldsymbol{r}}}{||\bar{\boldsymbol{r}}||^3}, \tag{161}$$

$$\frac{d^2\bar{\boldsymbol{x}}_B}{d\bar{t}^2} = \frac{1}{1+\alpha^{-1}}\frac{\bar{\boldsymbol{r}}}{||\bar{\boldsymbol{r}}||^3}\,. \tag{162}$$

In the limit $m_A \ll m_B$, i.e., $\alpha \ll 1$, object B stands still, say $\bar{\boldsymbol{x}}_B = 0$, and object A orbits according to

$$\frac{d^2\bar{\boldsymbol{x}}_A}{d\bar{t}^2} = -\frac{\bar{\boldsymbol{x}}_A}{||\bar{\boldsymbol{x}}_A||^3}\,.$$

Solution in a special case: planet orbiting a star. To better see the motion, and that our scaling is reasonable, we introduce polar coordinates r and θ:

$$\bar{\boldsymbol{x}}_A = r\cos\theta\boldsymbol{i} + r\sin\theta\boldsymbol{j},$$

which means $\bar{\boldsymbol{x}}_A$ can be written as $\bar{\boldsymbol{x}}_A = r\boldsymbol{i}_r$. Since

$$\frac{d}{dt}\boldsymbol{i}_r = \dot{\theta}\boldsymbol{i}_\theta, \quad \frac{d}{dt}\boldsymbol{i}_\theta = -\dot{\theta}\boldsymbol{i}_r,$$

we have

$$\frac{d^2\bar{\boldsymbol{x}}_A}{d\bar{t}^2} = (\ddot{r} - r\dot{\theta}^2)\boldsymbol{i}_r + (r\ddot{\theta} + 2\dot{r}\dot{\theta})\boldsymbol{i}_\theta\,.$$

The equation of motion for mass A is then

$$\begin{aligned}\ddot{r} - r\dot{\theta}^2 &= -\frac{1}{r^2},\\ r\ddot{\theta} + 2\dot{r}\dot{\theta} &= 0\,.\end{aligned}$$

The special case of circular motion, $r = 1$, fulfills the equations, since the latter equation then gives $\dot{\theta} = \text{const}$ and the former then gives $\dot{\theta} = 1$, i.e., the motion is $r(t) = 1$, $\theta(t) = t$, with unit angular frequency as expected and period 2π as expected.

12.11 Electric circuits

Although the term "mechanical vibrations" is used in the present book, we must mention that the same type of equations arise when modeling electric circuits. The current $I(t)$ in a circuit with an inductor with inductance L, a capacitor with capacitance C, and overall resistance R, is governed by

$$\ddot{I} + \frac{R}{L}\dot{I} + \frac{1}{LC}I = \dot{V}(t), \tag{163}$$

where $V(t)$ is the voltage source powering the circuit. This equation has the same form as the general model considered in Section 10 if we set $u = I$, $f(u') = bu'$ and define $b = R/L$, $s(u) = L^{-1}C^{-1}u$, and $F(t) = \dot{V}(t)$.

13 Exercises

Exercise 22: Simulate resonance

We consider the scaled ODE model (126) from Section 12.2. After scaling, the amplitude of u will have a size about unity as time grows and the effect of the initial conditions die out due to damping. However, as $\gamma \to 1$, the amplitude of u increases, especially if β is small. This effect is called *resonance*. The purpose of this exercise is to explore resonance.

a) Figure out how the `solver` function in `vib.py` can be called for the scaled ODE (126).

Solution.

Comparing the scaled ODE (126) with the ODE (125) with dimensions, we realize that the parameters in the latter must be set as

- $m = 1$
- $f(\dot{u}) = 2\beta|\dot{u}|\dot{u}$
- $s(u) = ku$
- $F(t) = \sin(\gamma t)$
- $I = Ik/A$
- $V = \sqrt{mk}V/A$

The expected period is 2π, so simulating for N periods means $T = 2\pi N$. Having m time steps per period means $\Delta t = 2\pi/m$.

Suppose we just choose $I = 1$ and $V = 0$. Simulating for 20 periods with 60 time steps per period, implies the following `solver` call to run the scaled model:

```
u, t = solver(I=1, V=0, m=1, b=2*beta, s=lambda u: u,
              F=lambda t: sin(gamma*t), dt=2*pi/60,
              T=2*pi*20, damping='quadratic')
```

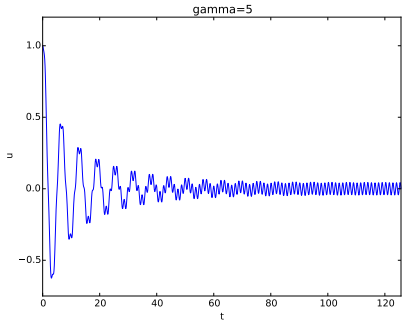
b) Run $\gamma = 5, 1.5, 1.1, 1$ for $\beta = 0.005, 0.05, 0.2$. For each β value, present an image with plots of $u(t)$ for the four γ values.

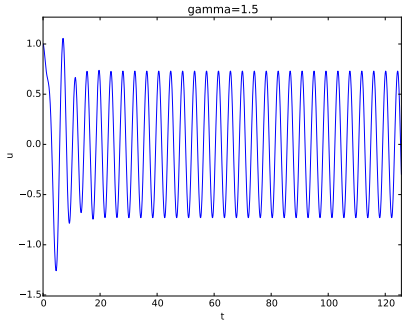
Solution.

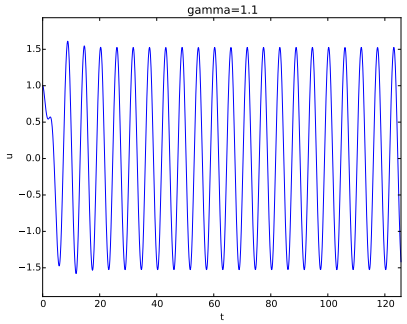
An appropriate program is

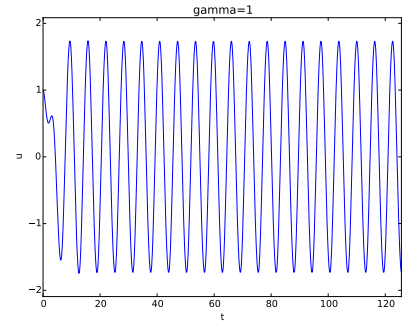
```
from vib import solver, visualize, plt
from math import pi, sin
import numpy as np

beta_values = [0.005, 0.05, 0.2]
beta_values = [0.00005]
gamma_values = [5, 1.5, 1.1, 1]
for i, beta in enumerate(beta_values):
    for gamma in gamma_values:
        u, t = solver(I=1, V=0, m=1, b=2*beta, s=lambda u: u,
                      F=lambda t: sin(gamma*t), dt=2*pi/60,
                      T=2*pi*20, damping='quadratic')
        visualize(u, t, title='gamma=%g' %
                  gamma, filename='tmp_%s' % gamma)
        print gamma, 'max u amplitude:', np.abs(u).max()
    for ext in 'png', 'pdf':
        cmd = 'doconce combine_images '
        cmd += ' '.join(['tmp_%s.' % gamma + ext
                         for gamma in gamma_values])
        cmd += ' resonance%d.' % (i+1) + ext
        os.system(cmd)
raw_input()
```

For $\beta = 0.2$ we see that the amplitude is not far from unity:

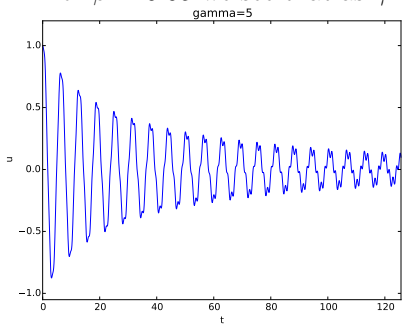


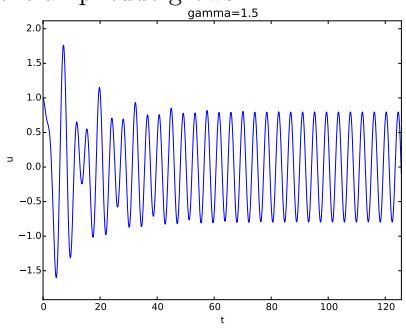


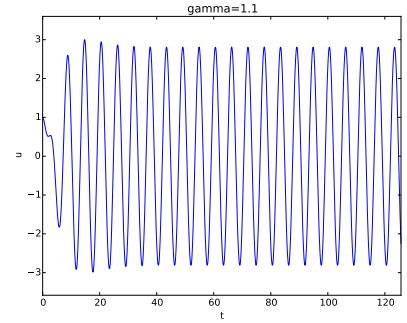


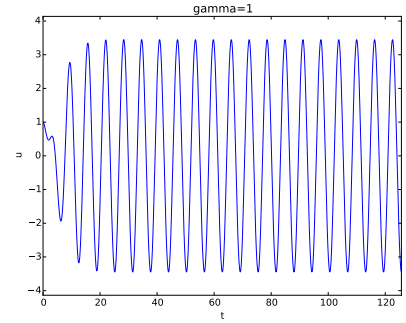


For $\beta = 0.05$ we see that as $\gamma \to 1$, the amplitude grows:

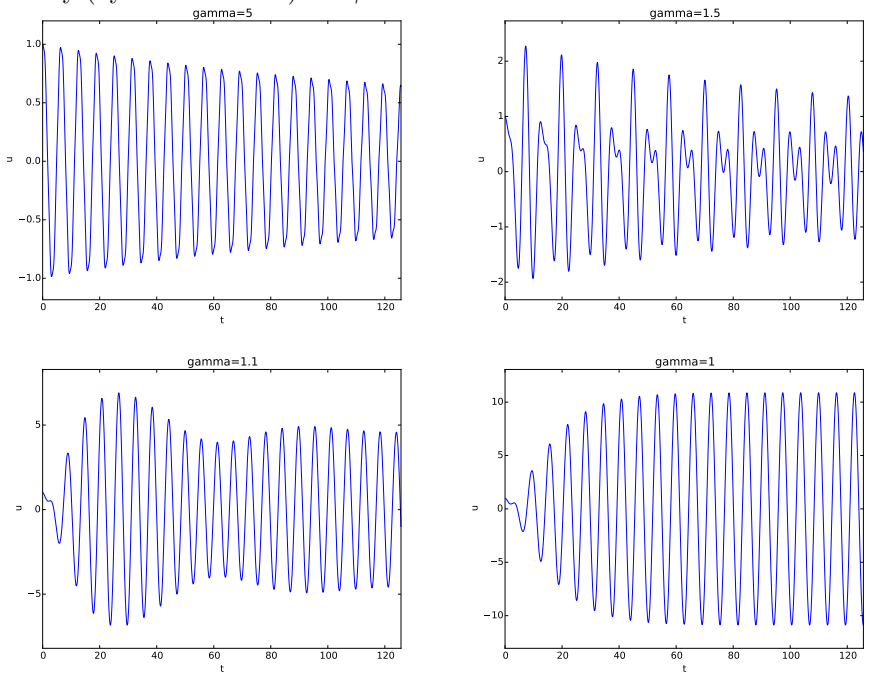








Finally, a small damping ($\beta = 0.005$) amplifies the amplitude significantly (by a factor of 10) for $\gamma = 1$:



For a very small $\beta = 0.00005$, the amplitude grows linearly up to about 60 for $\bar{t} \in [0, 120]$.

Filename: `resonance`.

Exercise 23: Simulate oscillations of a sliding box

Consider a sliding box on a flat surface as modeled in Section 12.3. As spring force we choose the nonlinear formula

$$s(u) = \frac{k}{\alpha}\tanh(\alpha u) = ku + \frac{1}{3}\alpha^2 ku^3 + \frac{2}{15}\alpha^4 ku^5 + \mathcal{O}(u^6)\,.$$

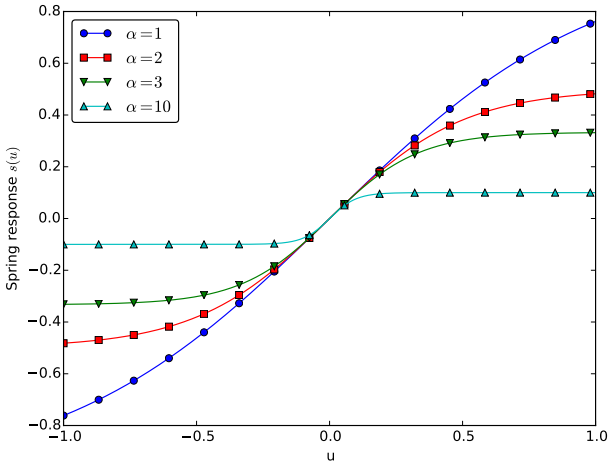
a) Plot $g(u) = \alpha^{-1}\tanh(\alpha u)$ for various values of α. Assume $u \in [-1, 1]$.

Solution.

Here is a function that does the plotting:

```
import scitools.std as plt
import numpy as np

def plot_spring():
```

```
alpha_values = [1, 2, 3, 10]
s = lambda u: 1.0/alpha*np.tanh(alpha*u)
u = np.linspace(-1, 1, 1001)
for alpha in alpha_values:
    print alpha, s(u)
    plt.plot(u, s(u))
    plt.hold('on')
plt.legend([r'$\alpha=%g$' % alpha for alpha in alpha_values])
plt.xlabel('u');  plt.ylabel('Spring response $s(u)$')
plt.savefig('tmp_s.png'); plt.savefig('tmp_s.pdf')
```



b) Scale the equations using I as scale for u and $\sqrt{m/k}$ as time scale.

Solution.

Inserting the dimensionless dependent and independent variables,

$$\bar{u} = \frac{u}{I}, \quad \bar{t} = \frac{t}{\sqrt{m/k}},$$

in the problem

$$m\ddot{u} + \mu mg\text{sign}(\dot{u}) + s(u) = 0, \quad u(0) = I,\ \dot{u}(0) = V,$$

gives

$$\frac{d^2\bar{u}}{d\bar{t}^2} + \frac{\mu mg}{kI}\text{sign}\left(\frac{d\bar{u}}{d\bar{t}}\right) + \frac{1}{\alpha I}\tanh(\alpha I\bar{u}) = 0, \quad \bar{u}(0) = 1,\ \frac{d\bar{u}}{d\bar{t}}(0) = \frac{V\sqrt{mk}}{kI}\,.$$

We can now identify three dimensionless parameters,

$$\beta = \frac{\mu m g}{kI}, \quad \gamma = \alpha I, \quad \delta = \frac{V\sqrt{mk}}{kI}\,.$$

The scaled problem can then be written

$$\frac{d^2\bar{u}}{d\bar{t}^2} + \beta \text{sign}\left(\frac{d\bar{u}}{d\bar{t}}\right) + \gamma^{-1}\tanh(\gamma\bar{u}) = 0, \quad \bar{u}(0) = 1,\ \frac{d\bar{u}}{d\bar{t}}(0) = \delta\,.$$

The initial set of 7 parameters $(\mu, m, g, k, \alpha, I, V)$ are reduced to 3 dimensionless combinations.

c) Implement the scaled model in b). Run it for some values of the dimensionless parameters.

Solution.

We use Odespy to solve the ODE, which requires rewriting the ODE as a system of two first-order ODEs:

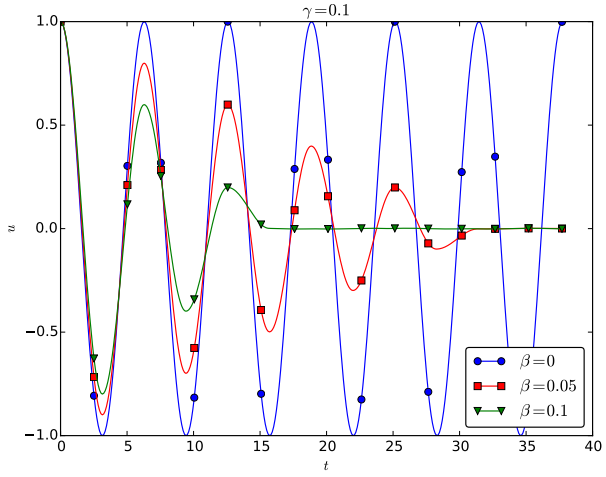
$$v' = -\beta\text{sign}(v) - \gamma^{-1}\tanh(\gamma\bar{u}),$$
$$u' = v,$$

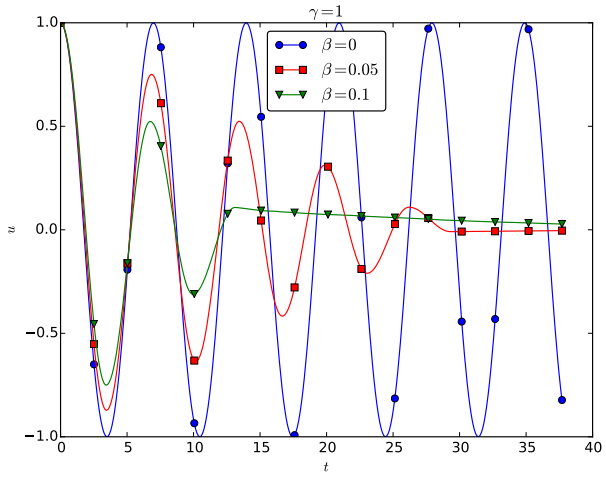
with initial conditions $v(0) = \delta$ and $u(0) = 1$. Here, $u(t)$ corresponds to the previous $\bar{u}(\bar{t})$, while $v(t)$ corresponds to $d\bar{u}/d\bar{t}(\bar{t})$. The code can be like this:

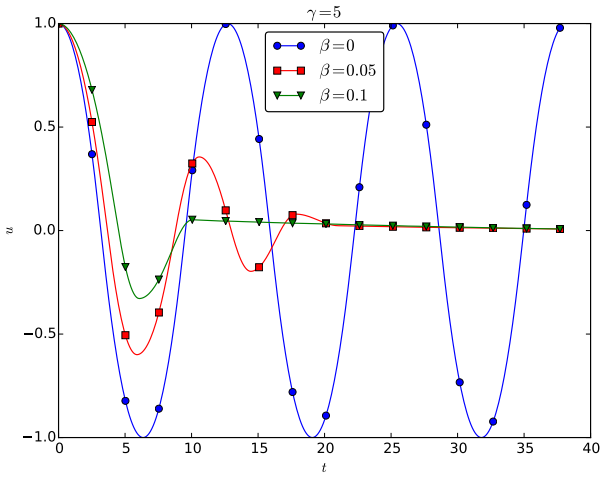
```
def simulate(beta, gamma, delta=0,
             num_periods=8, time_steps_per_period=60):
    # Use oscillations without friction to set dt and T
    P = 2*np.pi
    dt = P/time_steps_per_period
    T = num_periods*P
    t = np.linspace(0, T, time_steps_per_period*num_periods+1)
    import odespy
    def f(u, t, beta, gamma):
        # Note the sequence of unknowns: v, u (v=du/dt)
        v, u = u
        return [-beta*np.sign(v) - 1.0/gamma*np.tanh(gamma*u), v]
        #return [-beta*np.sign(v) - u, v]

    solver = odespy.RK4(f, f_args=(beta, gamma))
    solver.set_initial_condition([delta, 1]) # sequence must match f
    uv, t = solver.solve(t)
```

```
    u = uv[:,1]  # recall sequence in f: v, u
    v = uv[:,0]
    return u, t
```

We simulate for an almost linear spring in the regime of $\bar{u}$ (recall that $\bar{u} \in [0,1]$ since u is scaled with I), which corresponds to $\alpha = 1$ in a) and therefore $\gamma = 1$. Then we can try a spring whose force quickly flattens out like $\alpha = 5$ in a), which corresponds to $\gamma = 5$ in the scaled model. A third option is to have a truly linear spring, e.g., $\gamma = 0.1$. After some experimentation we realize that $\beta = 0, 0.05, 0.1$ are relevant values.







Filename: `sliding_box`.

Exercise 24: Simulate a bouncing ball

Section 12.9 presents a model for a bouncing ball. Choose one of the two ODE formulation, (156) or (157)-(158), and simulate the motion of a bouncing ball. Plot $h(t)$. Think about how to plot $v(t)$.

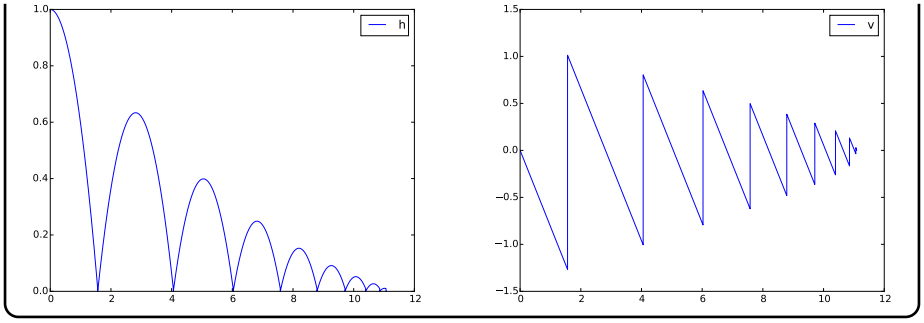
Hint. A naive implementation may get stuck in repeated impacts for large time step sizes. To avoid this situation, one can introduce a state variable that holds the mode of the motion: free fall, impact, or rest. Two consecutive impacts imply that the motion has stopped.

Solution.

A tailored `solver` function and some plotting statements go like

```
import numpy as np

def solver(H, C_R, dt, T, eps_v=0.01, eps_h=0.01):
    """
    Simulate bouncing ball until it comes to rest. Time step dt.
    h(0)=H (initial height). T: maximum simulation time.
    Method: Euler-Cromer.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    h = np.zeros(Nt+1)
```

```
    v = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)
    g = 0.81

    v[0] = 0
    h[0] = H
    mode = 'free fall'
    for n in range(Nt):
        v[n+1] = v[n] - dt*g
        h[n+1] = h[n] + dt*v[n+1]

        if h[n+1] < eps_h:
            #if abs(v[n+1]) > eps_v:  # handles large dt, but is wrong
            if v[n+1] < -eps_v:
                # Impact
                v[n+1] = -C_R*v[n+1]
                h[n+1] = 0
                if mode == 'impact':
                    # impact twice
                    return h[:n+2], v[:n+2], t[:n+2]
                mode = 'impact'
            elif abs(v[n+1]) < eps_v:
                mode = 'rest'
                v[n+1] = 0
                h[n+1] = 0
                return h[:n+2], v[:n+2], t[:n+2]
            else:
                mode = 'free fall'
        else:
            mode = 'free fall'
        print '%4d v=%8.5f h=%8.5f %s' % (n, v[n+1], h[n+1], mode)
    raise ValueError('T=%g is too short simulation time' % T)

import matplotlib.pyplot as plt
h, v, t = solver(
    H=1, C_R=0.8, T=100, dt=0.0001, eps_v=0.01, eps_h=0.01)
plt.plot(t, h)
plt.legend('h')
plt.savefig('tmp_h.png'); plt.savefig('tmp_h.pdf')
plt.figure()
plt.plot(t, v)
plt.legend('v')
plt.savefig('tmp_v.png'); plt.savefig('tmp_v.pdf')
plt.show()
```



Filename: `bouncing_ball`.

Exercise 25: Simulate a simple pendulum

Simulation of simple pendulum can be carried out by using the mathematical model derived in Section 12.5 and calling up functionality in the `vib.py` file (i.e., solve the second-order ODE by centered finite differences).

a) Scale the model. Set up the dimensionless governing equation for θ and expressions for dimensionless drag and wire forces.

Solution.

The angle is measured in radians so we may think of this quantity as dimensionless, or we may scale it by the initial condition to obtain a primary unknown that lies in $[-1, 1]$. We go for the former strategy here.

Dimensionless time $\bar{t}$ is introduced as t/t_c for some suitable time scale t_c. Inserted in the two governing equations (130) and (128), for the two unknowns θ and S, respectively, we achieve

$$-S + mg\cos\theta = -\frac{1}{t_c}L\frac{d\theta}{d\bar{t}},$$

$$\frac{1}{t_c^2}m\frac{d^2\theta}{d\bar{t}^2} + \frac{1}{2}C_D\varrho AL\frac{1}{t_c^2}\left|\frac{d\theta}{d\bar{t}}\right|\frac{d\theta}{d\bar{t}} + \frac{mg}{L}\sin\theta = 0\,.$$

We multiply the latter equation by t_c^2/m to make each term dimensionless:

$$\frac{d^2\theta}{d\bar{t}^2} + \frac{1}{2m}C_D\varrho AL\left|\frac{d\theta}{d\bar{t}}\right|\frac{d\theta}{d\bar{t}} + \frac{t_c^2 g}{L}\sin\theta = 0\,.$$

Assuming that the acceleration term and the gravity term to be the dominating terms, these should balance, so $t_c^2 g/L = 1$, giving $t_c = \sqrt{g/L}$. With $A = \pi R^2$ we get the dimensionless ODEs

$$\frac{d^2\theta}{d\bar{t}^2} + \alpha\left|\frac{d\theta}{d\bar{t}}\right|\frac{d\theta}{d\bar{t}} + \sin\theta = 0, \tag{164}$$

$$\frac{S}{mg} = \left(\frac{d\theta}{d\bar{t}}\right)^2 + \cos\theta, \tag{165}$$

where α is a dimensionless drag coefficient

$$\alpha = \frac{C_D\varrho\pi R^2 L}{2m}\,.$$

Note that in (165) we have divided by mg, which is in fact a force scale, making the gravity force unity and also $S/mg = 1$ in the equilibrium position $\theta = 0$. We may introduce

$$\bar{S} = S/mg$$

as a dimensionless drag force.

The parameter α is about the ratio of the drag force and the gravity force:

$$\frac{|\frac{1}{2}C_D\varrho\pi R^2|v|v|}{|mg|} \sim \frac{C_D\varrho\pi R^2L^2t_c^{-2}}{mg}\left|\frac{d\bar{\theta}}{d\bar{t}}\right|\frac{d\bar{\theta}}{d\bar{t}} \sim \frac{C_D\varrho\pi R^2 L}{2m}\Theta^2 = \alpha\Theta^2\,.$$

(We have that $\theta(t)/d\Theta$ is in $[-1,1]$, so we expect since $\Theta^{-1}d\bar{\theta}/d\bar{t}$ to be around unity. Here, $\Theta = \theta(0)$.)

Let us introduce ω for the dimensionless angular velocity,

$$\omega = \frac{d\theta}{d\bar{t}}\,.$$

When θ is computed, the dimensionless wire and drag forces are computed by

$$\bar{S} = \omega^2 + \cos\theta,$$
$$\bar{D} = -\alpha|\omega|\omega\,.$$

b) Write a function for computing θ and the dimensionless drag force and the force in the wire, using the `solver` function in the `vib.py` file. Plot these three quantities below each other (in subplots) so the graphs can be compared. Run two cases, first one in the limit of Θ small and no drag, and then a second one with $\Theta = 40$ degrees and $\alpha = 0.8$.

Solution.

The first step is to realize how to utilize the `solver` function for our dimensionless model. Introducing `Theta` for Θ, the arguments to `solver` must be set as

```
I = Theta
V = 0
m = 1
b = alpha
s = lambda u: sin(u)
F = lambda t: 0
damping = 'quadratic'
```

After computing θ, we need to find ω by finite differences:

$$\omega^n = \frac{\theta^{n+1} - \theta^{n-1}}{2\Delta t},\ n = 1, \ldots, N_t - 1, \quad \omega^0 = \frac{\theta^1 - \theta^0}{\Delta t},\ \omega^{N_t} = \frac{\theta^{N_t} - \theta^{N_t - 1}}{\Delta t}\,.$$

The duration of the simulation and the time step can be computed on basis of the analytical insight we have for small θ ($\theta \approx \Theta\cos(t)$). A complete function then reads

```
def simulate(Theta, alpha, num_periods=10):
    # Dimensionless model requires the following parameters:
    from math import sin, pi

    I = Theta
    V = 0
    m = 1
    b = alpha
    s = lambda u: sin(u)
    F = lambda t: 0
    damping = 'quadratic'

    # Estimate T and dt from the small angle solution
    P = 2*pi   # One period (theta small, no drag)
    dt = P/40  # 40 intervals per period
    T = num_periods*P

    theta, t =  solver(I, V, m, b, s, F, dt, T, damping)
    omega = np.zeros(theta.size)
    omega[1:-1] = (theta[2:] - theta[:-2])/(2*dt)
    omega[0] = (theta[1] - theta[0])/dt
    omega[-1] = (theta[-1] - theta[-2])/dt

    S = omega**2 + np.cos(theta)
    D = alpha*np.abs(omega)*omega
```

```
    return t, theta, S, D
```

Assuming imports like

```
import numpy as np
import matplotlib.pyplot as plt
```

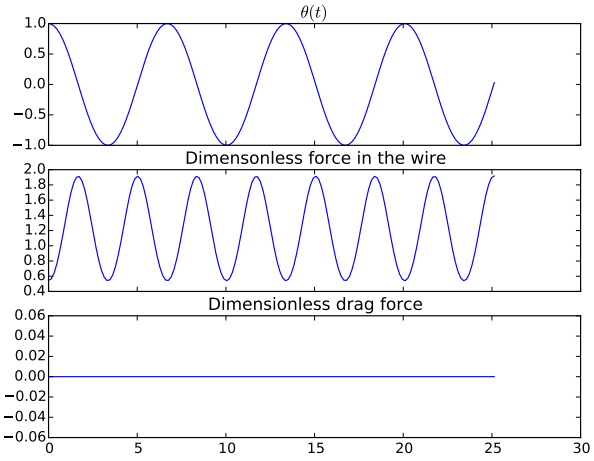
the following function visualizes θ, $\bar{S}$, and $\bar{D}$ with three subplots:

```
def visualize(t, theta, S, D, filename='tmp'):
    f, (ax1, ax2, ax3) = plt.subplots(3, sharex=True, sharey=False)
    ax1.plot(t, theta)
    ax1.set_title(r'$\theta(t)$')
    ax2.plot(t, S)
    ax2.set_title(r'Dimensonless force in the wire')
    ax3.plot(t, D)
    ax3.set_title(r'Dimensionless drag force')
    plt.savefig('%s.png' % filename)
    plt.savefig('%s.pdf' % filename)
```

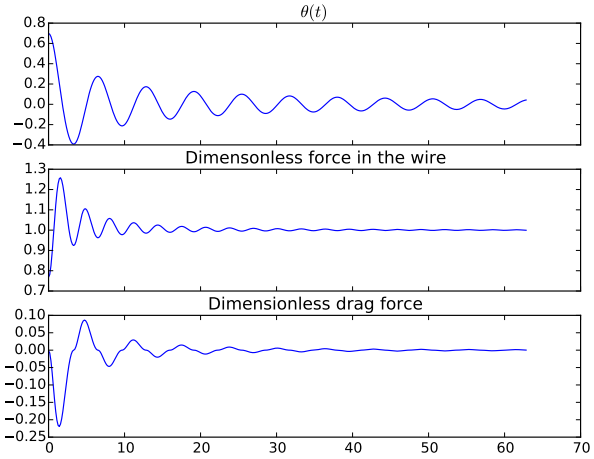
A suitable main program is

```
import math
# Rough verification that small theta and no drag gives cos(t)
Theta = 1.0
alpha = 0
t, theta, S, D = simulate(Theta, alpha, num_periods=4)
# Scale theta by Theta (easier to compare with cos(t))
theta /= Theta
visualize(t, theta, S, D, filename='pendulum_verify')

Theta = math.radians(40)
alpha = 0.8
t, theta, S, D = simulate(Theta, alpha)
visualize(t, theta, S, D, filename='pendulum_alpha0.8_Theta40')
plt.show()
```

The "verification" case looks good (at least when the `solver` function has been thoroughly verified in other circumstances):



The "real case" shows how quickly the drag force is reduced, even when we set α to a significant value (0.8):



Filename: `simple_pendulum`.

Exercise 26: Simulate an elastic pendulum

Section 12.7 describes a model for an elastic pendulum, resulting in a system of two ODEs. The purpose of this exercise is to implement the scaled model, test the software, and generalize the model.

a) Write a function `simulate` that can simulate an elastic pendulum using the scaled model. The function should have the following arguments:

```
def simulate(
    beta=0.9,                 # dimensionless parameter
    Theta=30,                 # initial angle in degrees
    epsilon=0,                # initial stretch of wire
    num_periods=6,            # simulate for num_periods
    time_steps_per_period=60, # time step resolution
    plot=True,                # make plots or not
    ):
```

To set the total simulation time and the time step, we use our knowledge of the scaled, classical, non-elastic pendulum: $u'' + u = 0$, with solution $u = \Theta \cos \bar{t}$. The period of these oscillations is $P = 2\pi$ and the frequency is unity. The time for simulation is taken as `num_periods` times P. The time step is set as P divided by `time_steps_per_period`.

The `simulate` function should return the arrays of x, y, θ, and t, where $\theta = \tan^{-1}(x/(1-y))$ is the angular displacement of the elastic pendulum corresponding to the position (x, y).

If `plot` is `True`, make a plot of $\bar{y}(\bar{t})$ versus $\bar{x}(\bar{t})$, i.e., the physical motion of the mass at $(\bar{x}, \bar{y})$. Use the equal aspect ratio on the axis such that we get a physically correct picture of the motion. Also make a plot of $\theta(\bar{t})$, where θ is measured in degrees. If $\Theta < 10$ degrees, add a plot that compares the solutions of the scaled, classical, non-elastic pendulum and the elastic pendulum ($\theta(t)$).

Although the mathematics here employs a bar over scaled quantities, the code should feature plain names `x` for $\bar{x}$, `y` for $\bar{y}$, and `t` for $\bar{t}$ (rather than `x_bar`, etc.). These variable names make the code easier to read and compare with the mathematics.

Hint 1. Equal aspect ratio is set by `plt.gca().set_aspect('equal')` in Matplotlib (`import matplotlib.pyplot as plt`) and in SciTools by the command `plt.plot(..., daspect=[1,1,1], daspectmode='equal')` (provided you have done `import scitools.std as plt`).

Hint 2. If you want to use Odespy to solve the equations, order the ODEs like $\dot{\bar{x}}, \bar{x}, \dot{\bar{y}}, \bar{y}$ such that `odespy.EulerCromer` can be applied.

Solution.

Here is a suggested `simulate` function:

```
import odespy
import numpy as np
import scitools.std as plt

def simulate(
    beta=0.9,                 # dimensionless parameter
    Theta=30,                 # initial angle in degrees
```

```
epsilon=0,                # initial stretch of wire
num_periods=6,            # simulate for num_periods
time_steps_per_period=60, # time step resolution
plot=True,                # make plots or not
):
from math import sin, cos, pi
Theta = Theta*np.pi/180  # convert to radians
# Initial position and velocity
# (we order the equations such that Euler-Cromer in odespy
# can be used, i.e., vx, x, vy, y)
ic = [0,                                # x'=vx
      (1 + epsilon)*sin(Theta),         # x
      0,                                # y'=vy
      1 - (1 + epsilon)*cos(Theta),     # y
      ]

def f(u, t, beta):
    vx, x, vy, y = u
    L = np.sqrt(x**2 + (y-1)**2)
    h = beta/(1-beta)*(1 - beta/L)  # help factor
    return [-h*x, vx, -h*(y-1) - beta, vy]

# Non-elastic pendulum (scaled similarly in the limit beta=1)
# solution Theta*cos(t)
P = 2*pi
dt = P/time_steps_per_period
T = num_periods*P
omega = 2*pi/P

time_points = np.linspace(
    0, T, num_periods*time_steps_per_period+1)

solver = odespy.EulerCromer(f, f_args=(beta,))
solver.set_initial_condition(ic)
u, t = solver.solve(time_points)
x = u[:,1]
y = u[:,3]
theta = np.arctan(x/(1-y))

if plot:
    plt.figure()
    plt.plot(x, y, 'b-', title='Pendulum motion',
             daspect=[1,1,1], daspectmode='equal',
             axis=[x.min(), x.max(), 1.3*y.min(), 1])
    plt.savefig('tmp_xy.png')
    plt.savefig('tmp_xy.pdf')
    # Plot theta in degrees
    plt.figure()
```

```
        plt.plot(t, theta*180/np.pi, 'b-',
                 title='Angular displacement in degrees')
        plt.savefig('tmp_theta.png')
        plt.savefig('tmp_theta.pdf')
        if abs(Theta) < 10*pi/180:
            # Compare theta and theta_e for small angles (<10 degrees)
            theta_e = Theta*np.cos(omega*t)  # non-elastic scaled sol.
            plt.figure()
            plt.plot(t, theta, t, theta_e,
                     legend=['theta elastic', 'theta non-elastic'],
                     title='Elastic vs non-elastic pendulum, '\
                           'beta=%g' % beta)
            plt.savefig('tmp_compare.png')
            plt.savefig('tmp_compare.pdf')
        # Plot y vs x (the real physical motion)
    return x, y, theta, t
```

b) Write a test function for testing that $\Theta = 0$ and $\epsilon = 0$ gives $x = y = 0$ for all times.

Solution.

Here is the code:

```
def test_equilibrium():
    """Test that starting from rest makes x=y=theta=0."""
    x, y, theta, t = simulate(
        beta=0.9, Theta=0, epsilon=0,
        num_periods=6, time_steps_per_period=10, plot=False)
    tol = 1E-14
    assert np.abs(x.max()) < tol
    assert np.abs(y.max()) < tol
    assert np.abs(theta.max()) < tol
```

c) Write another test function for checking that the pure vertical motion of the elastic pendulum is correct. Start with simplifying the ODEs for pure vertical motion and show that $\bar{y}(\bar{t})$ fulfills a vibration equation with frequency $\sqrt{\beta/(1-\beta)}$. Set up the exact solution.

Write a test function that uses this special case to verify the `simulate` function. There will be numerical approximation errors present in the results from `simulate` so you have to believe in correct results and set a (low) tolerance that corresponds to the computed maximum error. Use a small Δt to obtain a small numerical approximation error.

Solution.

For purely vertical motion, the ODEs reduce to $\ddot{x} = 0$ and

$$\frac{d^2\bar{y}}{d\bar{t}^2} = -\frac{\beta}{1-\beta}(1 - \beta\frac{1}{\sqrt{(\bar{y}-1)^2}})(\bar{y}-1) - \beta = -\frac{\beta}{1-\beta}(\bar{y} - 1 + \beta) - \beta\,.$$

We have here used that $(\bar{y}-1)/\sqrt{(\bar{y}-1)^2} = -1$ since $\bar{y}$ cannot exceed 1 (the pendulum's wire is fixed at the scaled point $(0, 1)$). In fact, $\bar{y}$ will be around zero. (As a consistency check, we realize that in equilibrium, $\ddot{\bar{y}} = 0$, and multiplying by $(1-\beta)/\beta$ leads to the expected $\bar{y} = 0$.) Further calculations easily lead to

$$\frac{d^2\bar{y}}{d\bar{t}^2} = -\frac{\beta}{1-\beta}\bar{y} = -\omega^2\bar{y},$$

where we have introduced the frequency $\omega = \sqrt{\beta/(1-\beta)}$. Solving this standard ODE, with an initial stretching $\bar{y}(0) = \epsilon$ and no velocity, results in

$$\bar{y}(\bar{t}) = \epsilon\cos(\omega\bar{t})\,.$$

Note that the oscillations we describe here are very different from the oscillations used to set the period and time step in function `simulate`. The latter type of oscillations are due to gravity when a classical, non-elastic pendulum oscillates back and forth, while $\bar{y}(\bar{t})$ above refers to vertical *elastic* oscillations in the wire around the equilibrium point in the gravity field. The angular frequency of the vertical oscillations are given by ω and the corresponding period is $\hat{P} = 2\pi/\omega$. Suppose we want to simulate for $T = N\hat{P} = N2\pi/\omega$ and use n time steps per period, $\Delta\bar{t} = \hat{P}/n$. The `simulate` function operates with a simulation time of `num_periods` times 2π. This means that we must set `num_periods=N/omega` if we want to simulate to time $T = N\hat{P}$. The parameter `time_steps_per_period` must be set to ωn since `simulate` has Δt as 2π divided by `time_steps_per_period` and we want $\Delta t = 2\pi\omega^{-1}n^{-1}$.

The corresponding test function can be written as follows.

```
def test_vertical_motion():
    beta = 0.9
    omega = np.sqrt(beta/(1-beta))
    # Find num_periods. Recall that P=2*pi for scaled pendulum
    # oscillations, while here we don't have gravity driven
    # oscillations, but elastic oscillations with frequency omega.
    period = 2*np.pi/omega
    # We want T = N*period
    N = 5
```

```
    # simulate function has T = 2*pi*num_periods
    num_periods = 5/omega
    n = 600
    time_steps_per_period = omega*n

    y_exact = lambda t: -0.1*np.cos(omega*t)
    x, y, theta, t = simulate(
        beta=beta, Theta=0, epsilon=0.1,
        num_periods=num_periods,
        time_steps_per_period=time_steps_per_period,
        plot=False)

    tol = 0.00055 # ok tolerance for the above resolution
    # No motion in x direction is epxected
    assert np.abs(x.max()) < tol
    # Check motion in y direction
    y_e = y_exact(t)
    diff = np.abs(y_e - y).max()
    if diff > tol: # plot
        plt.plot(t, y, t, y_e, legend=['y', 'exact'])
        raw_input('Error in test_vertical_motion; type CR:')
    assert diff < tol, 'diff=%g' % diff
```

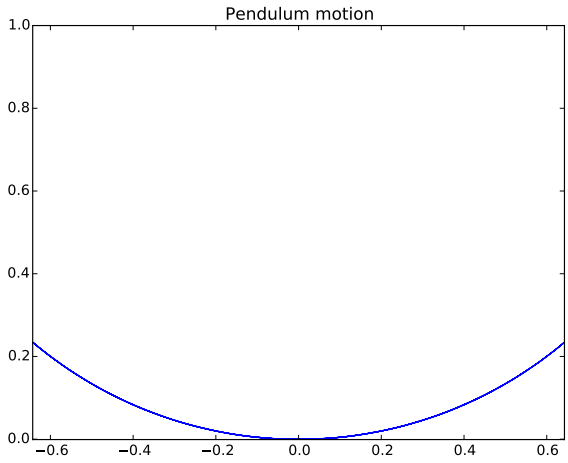
d) Make a function `demo(beta, Theta)` for simulating an elastic pendulum with a given β parameter and initial angle Θ. Use 600 time steps per period to get every accurate results, and simulate for 3 periods.

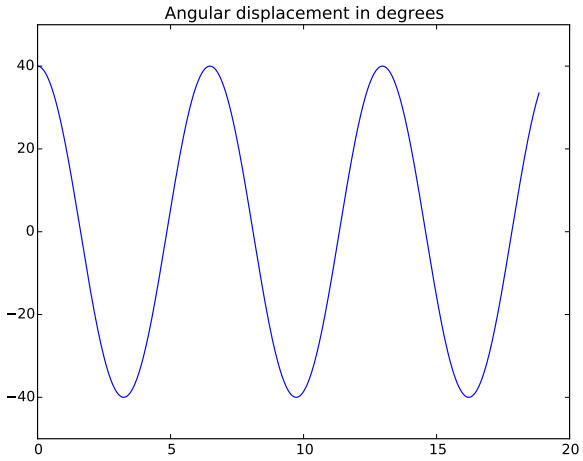
Solution.

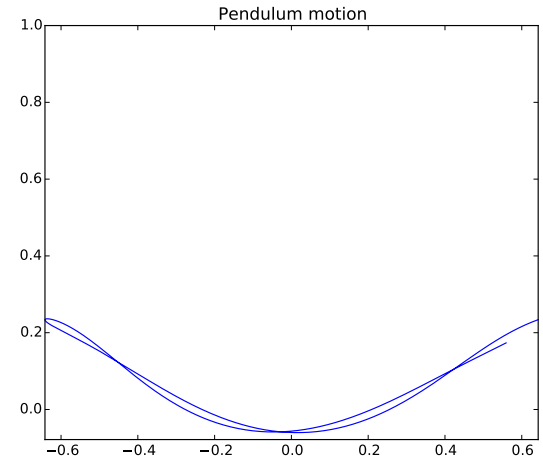
The `demo` function is just

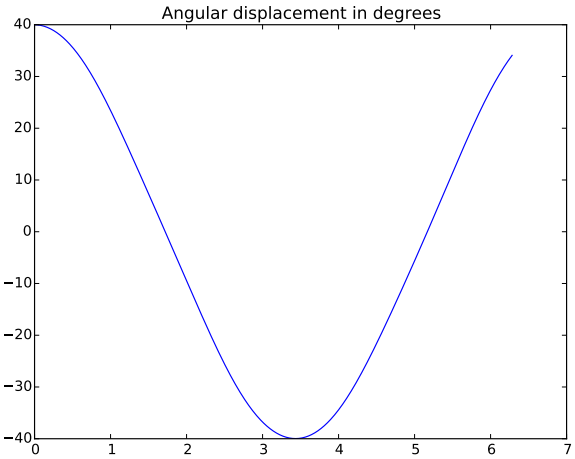
```
def demo(beta=0.999, Theta=40, num_periods=3):
    x, y, theta, t = simulate(
        beta=beta, Theta=Theta, epsilon=0,
        num_periods=num_periods, time_steps_per_period=600,
        plot=True)
```

Below are plots corresponding to $\beta = 0.999$ (3 periods) and $\beta = 0.93$ (one period):


Pendulum motion
1.0
0.8
0.6
0.4
0.2
0.0
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6


Angular displacement in degrees
40
20
0
−20
−40
0
5
10
15
20





Filename: `elastic_pendulum`.

Exercise 27: Simulate an elastic pendulum with air resistance

This is a continuation Exercise 26. Air resistance on the body with mass m can be modeled by the force $-\frac{1}{2}\varrho C_D A|\boldsymbol{v}|\boldsymbol{v}$, where C_D is a drag coefficient (0.2 for a sphere), ϱ is the density of air (1.2 kg m^{-3}), A is the cross section area ($A = \pi R^2$ for a sphere, where R is the radius), and $\boldsymbol{v}$ is the velocity of the body. Include air resistance in the original model, scale the model, write a function `simulate_drag` that is a copy of the `simulate` function from Exercise 26, but

with the new ODEs included, and show plots of how air resistance influences the motion.

Solution.

We start with the model (140)-(146). Since $\boldsymbol{v} = \dot{x}\boldsymbol{i} + \dot{y}\boldsymbol{j}$, the air resistance term can be written

$$-q(\dot{x}\boldsymbol{i} + \dot{y}\boldsymbol{j}), \quad q = \frac{1}{2}\varrho C_D A\sqrt{\dot{x}^2 + \dot{y}^2}\,.$$

Note that for positive velocities, the pendulum is moving to the right and the air resistance works against the motion, i.e., in direction of $-\boldsymbol{v} = -\dot{x}\boldsymbol{i} - \dot{y}\boldsymbol{j}$.

We can easily include the terms in the ODEs:

$$\ddot{x} = -\frac{q}{m}\dot{x} - \frac{k}{m}\left(1 - \frac{L_0}{L}\right)(x - x_0), \tag{166}$$

$$\ddot{y} = -\frac{q}{m}\dot{y} - \frac{k}{m}\left(1 - \frac{L_0}{L}\right)(y - y_0) - g, \tag{167}$$

$$L = \sqrt{(x - x_0)^2 + (y - y_0)^2}, \tag{168}$$

$$\tag{169}$$

The initial conditions are not affected.

The next step is to scale the model. We use the same scales as in Exercise 26, introduce β, and $A = \pi R^2$ to simplify the $-q\dot{x}/m$ term to

$$\frac{L_0}{2m}\varrho C_D R^2 \beta^{-1}\sqrt{\left(\frac{d\bar{x}}{d\bar{t}}\right)^2 + \left(\frac{d\bar{y}}{d\bar{t}}\right)^2} = \gamma\beta^{-1}\sqrt{\left(\frac{d\bar{x}}{d\bar{t}}\right)^2 + \left(\frac{d\bar{y}}{d\bar{t}}\right)^2},$$

where γ is a second dimensionless parameter:

$$\gamma = \frac{L_0}{2m}\varrho C_D R^2\,.$$

The final set of scaled equations is then

$$\frac{d^2\bar{x}}{d\bar{t}^2} = -\gamma\beta^{-1}\sqrt{\left(\frac{d\bar{x}}{d\bar{t}}\right)^2 + \left(\frac{d\bar{y}}{d\bar{t}}\right)^2}\frac{d\bar{x}}{d\bar{t}} - \frac{\beta}{1-\beta}\left(1 - \frac{\beta}{\bar{L}}\right)\bar{x}, \tag{170}$$

$$\frac{d^2\bar{y}}{d\bar{t}^2} = -\gamma\beta^{-1}\sqrt{\left(\frac{d\bar{x}}{d\bar{t}}\right)^2 + \left(\frac{d\bar{y}}{d\bar{t}}\right)^2}\frac{d\bar{y}}{d\bar{t}} - \frac{\beta}{1-\beta}\left(1 - \frac{\beta}{\bar{L}}\right)(\bar{y} - 1) - \beta, \tag{171}$$

$$\bar{L} = \sqrt{\bar{x}^2 + (\bar{y} - 1)^2}, \tag{172}$$

$$\bar{x}(0) = (1 + \epsilon)\sin\Theta, \tag{173}$$

$$\frac{d\bar{x}}{d\bar{t}}(0) = 0, \tag{174}$$

$$\bar{y}(0) = 1 - (1 + \epsilon)\cos\Theta, \tag{175}$$

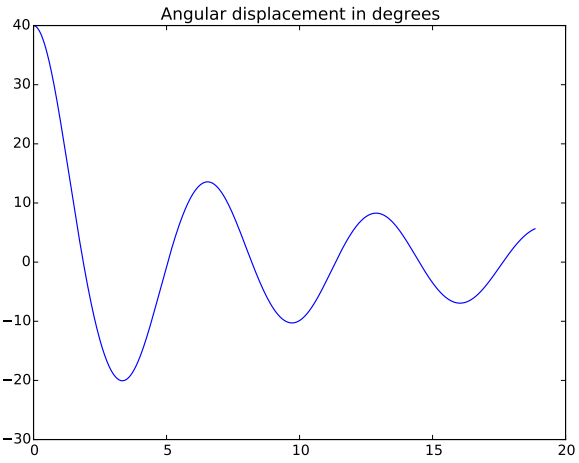
$$\frac{d\bar{y}}{d\bar{t}}(0) = 0, \tag{176}$$

The new `simulate_drag` function is implemented below.

```
def simulate_drag(
    beta=0.9,                 # dimensionless elasticity parameter
    gamma=0,                  # dimensionless drag parameter
    Theta=30,                 # initial angle in degrees
    epsilon=0,                # initial stretch of wire
    num_periods=6,            # simulate for num_periods
    time_steps_per_period=60, # time step resolution
    plot=True,                # make plots or not
    ):
    from math import sin, cos, pi
    Theta = Theta*np.pi/180  # convert to radians
    # Initial position and velocity
    # (we order the equations such that Euler-Cromer in odespy
    # can be used, i.e., vx, x, vy, y)
    ic = [0,                                  # x'=vx
          (1 + epsilon)*sin(Theta),           # x
          0,                                  # y'=vy
          1 - (1 + epsilon)*cos(Theta),       # y
          ]

    def f(u, t, beta, gamma):
        vx, x, vy, y = u
        L = np.sqrt(x**2 + (y-1)**2)
        v = np.sqrt(vx**2 + vy**2)
        h1 = beta/(1-beta)*(1 - beta/L)  # help factor
        h2 = gamma/beta*v
        return [-h2*vx - h1*x, vx, -h2*vy - h1*(y-1) - beta, vy]
```

```
# Non-elastic pendulum (scaled similarly in the limit beta=1)
# solution Theta*cos(t)
P = 2*pi
dt = P/time_steps_per_period
T = num_periods*P
omega = 2*pi/P

time_points = np.linspace(
    0, T, num_periods*time_steps_per_period+1)

solver = odespy.EulerCromer(f, f_args=(beta, gamma))
solver.set_initial_condition(ic)
u, t = solver.solve(time_points)
x = u[:,1]
y = u[:,3]
theta = np.arctan(x/(1-y))

if plot:
    plt.figure()
    plt.plot(x, y, 'b-', title='Pendulum motion',
             daspect=[1,1,1], daspectmode='equal',
             axis=[x.min(), x.max(), 1.3*y.min(), 1])
    plt.savefig('tmp_xy.png')
    plt.savefig('tmp_xy.pdf')
    # Plot theta in degrees
    plt.figure()
    plt.plot(t, theta*180/np.pi, 'b-',
             title='Angular displacement in degrees')
    plt.savefig('tmp_theta.png')
    plt.savefig('tmp_theta.pdf')
    if abs(Theta) < 10*pi/180:
        # Compare theta and theta_e for small angles (<10 degrees)
        theta_e = Theta*np.cos(omega*t)  # non-elastic scaled sol.
        plt.figure()
        plt.plot(t, theta, t, theta_e,
                 legend=['theta elastic', 'theta non-elastic'],
                 title='Elastic vs non-elastic pendulum, '\
                       'beta=%g' % beta)
        plt.savefig('tmp_compare.png')
        plt.savefig('tmp_compare.pdf')
    # Plot y vs x (the real physical motion)
return x, y, theta, t
```

The plot of θ shows the damping ($\beta = 0.999$):



Test functions for equilibrium and vertical motion are also included. These are as in Exercise 27, except that they call `simulate_drag` instead of `simulate`.

Filename: `elastic_pendulum_drag`.

Remarks. Test functions are challenging to construct for the problem with air resistance. You can reuse the tests from Exercise 27 for `simulate_drag`, but these tests does not verify the new terms arising from air resistance.

Exercise 28: Implement the PEFRL algorithm

We consider the motion of a planet around a star (Section 12.10). The simplified case where one mass is very much bigger than the other and one object is at rest, results in the scaled ODE model

$$\ddot{x} + (x^2 + y^2)^{-3/2} x = 0,$$
$$\ddot{y} + (x^2 + y^2)^{-3/2} y = 0\,.$$

a) It is easy to show that $x(t)$ and $y(t)$ go like sine and cosine functions. Use this idea to derive the exact solution.

Solution.

We may assume $x = C_x \cos(\omega t)$ and $y = C_y \sin(\omega t)$ for constants C_x,, C_y, and ω. Inserted in the equations, we see that $\omega = 1$. The initial conditions determine the other constants, which we may choose as $C_x = C_y = 1$ (the

object starts at $(1, 0)$ with a velocity $(0, 1)$). The motion is a perfect circle, which should last forever.

b) One believes that a planet may orbit a star for billions of years. We are now interested in how accurate methods we actually need for such calculations. A first task is to determine what the time interval of interest is in scaled units. Take the earth and sun as typical objects and find the characteristic time used in the scaling of the equations ($t_c = \sqrt{L^3/(mG)}$), where m is the mass of the sun, L is the distance between the sun and the earth, and G is the gravitational constant. Find the scaled time interval corresponding to one billion years.

Solution.

According to Wikipedia, the mass of the sun is approximately $2 \cdot 10^{30}$ kg. This is 332946 times the mass of the earth, implying that the dimensionless constant $\alpha \approx 3 \cdot 10^{-6}$. With $G = 6.674 \cdot 10^{-11}$ Nm^2/kg^2, and the sun-earth distance as (approximately) 150 million km, we have $t_c \approx 5028388$ s. This is about 58 days, which is the characteristic time, chosen as the angular frequency of the oscillations. To get the period of one orbit we therefore must multiply by 2π. This gives about 1 year (and demonstrates the fact mentioned about the scaling: the natural time scale is consistent with Kepler's law about the period).

Thus, one billion years correspond to 62,715,924,070 time units (dividing one billion years by t_c), which corresponds to about 2000 "time unit years".

c) Solve the equations using 4th-order Runge-Kutta and the Euler-Cromer methods. You may benefit from applying Odespy for this purpose. With each solver, simulate 10000 orbits and print the maximum position error and CPU time as a function of time step. Note that the maximum position error does not necessarily occur at the end of the simulation. The position error achieved with each solver will depend heavily on the size of the time step. Let the time step correspond to 200, 400, 800 and 1600 steps per orbit, respectively. Are the results as expected? Explain briefly. When you develop your program, have in mind that it will be extended with an implementation of the other algorithms (as requested in d) and e) later) and experiments with this algorithm as well.

Solution.

The first task is to implement the right-hand side function for the system of ODEs such that we can call up Odespy solvers (or make use of other types of ODE software, e.g., from SciPy). The 2×2 system of second-order ODEs must be expressed as a 4×4 system of first-order ODEs. We have three different cases of right-hand sides:

1. Common numbering of unknowns: x, v_x, y, y_x
2. Numbering required by Euler-Cromer: v_x, x, v_y, y
3. Numbering required by the PEFRL method: same as Euler-Cromer

Most Odespy solvers can handle any convention for numbering of the unknowns. The important point is that initial conditions and new values at the end of the time step are filled in the right positions of a one-dimensional array containing the unknowns. Using Odespy to solve the system by the Euler-Cromer method, however, requires the unknowns to appear as velocity 1st degree-of-freedom, displacement 1st degree-of-freedom, velocity 2nd degree-of-freedom, displacement 2nd degree-of-freedom, and so forth. Two alternative right-hand side functions `f(u, t)` for Odespy solvers is then

```
def f_EC(u, t):
    '''
    Return derivatives for the 1st order system as
    required by Euler-Cromer.
    '''
    vx, x, vy, y = u  # u: array holding vx, x, vy, y
    d = -(x**2 + y**2)**(-3.0/2)
    return [d*x, vx, d*y, vy ]
def f_RK4(u, t):
    '''
    Return derivatives for the 1st order system as
    required by ordinary solvers in Odespy.
    '''
    x, vx, y, vy = u  # u: array holding x, vx, y, vy
    d = -(x**2 + y**2)**(-3.0/2)
    return [vx, d*x, vy, d*y ]
```

In addition, we shall later in d) implement the PEFRL method and just give the g function as input to a system of the form $dv_x = g_x$, $dv_y = g_y$, and g becomes the vector (g_x, g_y):

Some prefer to number the unknowns differently, and with the RK4 method we are free to use any numbering, including this one:

```
def g(u, v):
    return np.array([-u])
def u_exact(t):
    return np.array([3*np.cos(t)]).transpose()
I = u_exact(0)
V = np.array([0])
print 'V:', V, 'I:', I
# Numerical parameters
w = 1
```

```
P = 2*np.pi/w
dt_values = [P/20, P/40, P/80, P/160, P/320]
T = 8*P
error_vs_dt = []
for n, dt in enumerate(dt_values):
    u, v, t = solver_PEFRL(I, V, g, dt, T)
    error = np.abs(u - u_exact(t)).max()
    print 'error:', error
    if n > 0:
        error_vs_dt.append(error/dt**4)
for i in range(1, len(error_vs_dt)):
    #print abs(error_vs_dt[i]- error_vs_dt[0])
    assert abs(error_vs_dt[i]-
               error_vs_dt[0]) < 0.1
s PEFRL(odespy.Solver):
"""Class wrapper for Odespy."""  # Not used!
quick_desctiption = "Explicit 4th-order method for v'=-f, u=v."""
def advance(self):
    u, f, n, t = self.u, self.f, self.n, self.t
    dt = t[n+1] - t[n]
    I = np.array([u[1], u[3]])
    V = np.array([u[0], u[2]])
    u, v, t = solver_PFFRL(I, V, f, dt, t+dt)
    return np.array([v[-1], u[-1]])
compute_orbit_and_error(
f,
solver_ID,
timesteps_per_period=20,
N_orbit_groups=1000,
orbit_group_size=10):
'''
For one particular solver:
Calculte the orbits for a multiple of grouped orbits, i.e.
number of orbits = orbit_group_size*N_orbit_groups.
Returns: time step dt, and, for each N_orbit_groups cycle,
the 2D position error and cpu time (as lists).
'''
def u_exact(t):
    return np.array([np.cos(t), np.sin(t)])
w = 1
P = 2*np.pi/w       # scaled period (1 year becomes 2*pi)
dt = P/timesteps_per_period
Nt = orbit_group_size*N_orbit_groups*timesteps_per_period
T = Nt*dt
t_mesh = np.linspace(0, T, Nt+1)
E_orbit = []
#print '        dt:', dt
T_interval = P*orbit_group_size
```

```
N = int(round(T_interval/dt))
# set initial conditions
if solver_ID == 'EC':
    A = [0,1,1,0]
elif solver_ID == 'PEFRL':
    I = np.array([1, 0])
    V = np.array([0, 1])
else:
    A = [1,0,0,1]
t1 = time.clock()
for i in range(N_orbit_groups):
    time_points = np.linspace(i*T_interval, (i+1)*T_interval,N+1)
    u_e = u_exact(time_points).transpose()
    if solver_ID == 'EC':
        solver = odespy.EulerCromer(f)
        solver.set_initial_condition(A)
        ui, ti = solver.solve(time_points)
        # Find error (correct final pos:  x=1, y=0)
        orbit_error = np.sqrt(
            (ui[:,1]-u_e[:,0])**2 + (ui[:,3]-u_e[:,1])**2).max()
    elif solver_ID == 'PEFRL':
        # Note: every T_inverval is here counted from time 0
        ui, vi, ti = solver_PEFRL(I, V, f, dt, T_interval)
        # Find error (correct final pos:  x=1, y=0)
        orbit_error = np.sqrt(
            (ui[:,0]-u_e[:,0])**2 + (ui[:,1]-u_e[:,1])**2).max()
    else:
        solver = eval('odespy.' + solver_ID(f)
        solver.set_initial_condition(A)
        ui, ti = solver.solve(time_points)
        # Find error (correct final pos:  x=1, y=0)
        orbit_error = np.sqrt(
            (ui[:,0]-u_e[:,0])**2 + (ui[:,2]-u_e[:,1])**2).max()
    print '      Orbit no. %d,   max error (per cent): %g' % \
                        ((i+1)*orbit_group_size, orbit_error)
    E_orbit.append(orbit_error)
    # set init. cond. for next time interval
    if solver_ID == 'EC':
        A = [ui[-1,0], ui[-1,1], ui[-1,2], ui[-1,3]]
    elif solver_ID == 'PEFRL':
        I = [ui[-1,0], ui[-1,1]]
        V = [vi[-1,0], vi[-1,1]]
    else:  # RK4, adaptive rules, etc.
        A = [ui[-1,0], ui[-1,1], ui[-1,2], ui[-1,3]]
t2 = time.clock()
CPU_time = (t2 - t1)/(60.0*60.0)    # in hours
return dt, E_orbit, CPU_time
orbit_error_vs_dt(
```

```
f_EC, f_RK4, g, solvers,
N_orbit_groups=1000,
orbit_group_size=10):
'''
With each solver in list "solvers": Simulate
orbit_group_size*N_orbit_groups orbits with different dt values.
Collect final 2D position error for each dt and plot all errors.
'''
for solver_ID in solvers:
    print 'Computing orbit with solver:', solver_ID
    E_values = []
    dt_values = []
    cpu_values = []
    for timesteps_per_period in 200, 400, 800, 1600:
        print '.......time steps per period: ', \
                                timesteps_per_period
        if solver_ID == 'EC':
            dt, E, cpu_time = compute_orbit_and_error(
                f_EC,
                solver_ID,
                timesteps_per_period,
                N_orbit_groups,
                orbit_group_size)
        elif solver_ID == 'PEFRL':
            dt, E, cpu_time = compute_orbit_and_error(
                g,
                solver_ID,
                timesteps_per_period,
                N_orbit_groups,
                orbit_group_size)
        else:
            dt, E, cpu_time = compute_orbit_and_error(
                f_RK4,
                solver_ID,
                timesteps_per_period,
                N_orbit_groups,
                orbit_group_size)
        dt_values.append(dt)
        E_values.append(np.array(E).max())
        cpu_values.append(cpu_time)
    print 'dt_values:', dt_values
    print 'E max with dt...:', E_values
    print 'cpu_values with dt...:', cpu_values
orbit_error_vs_years(
f_EC, f_RK4, g, solvers,
N_orbit_groups=1000,
orbit_group_size=100,
N_time_steps = 1000):
```

```
'''
For each solver in the list solvers:
simulate orbit_group_size*N_orbit_groups orbits with a fixed
dt corresponding to N_time_steps steps per year.
Collect max 2D position errors for each N_time_steps'th run,
plot these errors and CPU. Finally, make an empirical
formula for error and CPU as functions of a number
of cycles.
'''
timesteps_per_period = N_time_steps     # fixed for all runs
for solver_ID in solvers:
    print 'Computing orbit with solver:', solver_ID
    if solver_ID == 'EC':
        dt, E, cpu_time = compute_orbit_and_error(
            f_EC,
            solver_ID,
            timesteps_per_period,
            N_orbit_groups,
            orbit_group_size)
    elif solver_ID == 'PEFRL':
        dt, E, cpu_time = compute_orbit_and_error(
            g,
            solver_ID,
            timesteps_per_period,
            N_orbit_groups,
            orbit_group_size)
    else:
        dt, E, cpu_time = compute_orbit_and_error(
            f_RK4,
            solver_ID,
            timesteps_per_period,
            N_orbit_groups,
            orbit_group_size)
    # E and cpu_time are for every N_orbit_groups cycle
    print 'E_values (fixed dt, changing no of years):', E
    print 'CPU (hours):', cpu_time
    years = np.arange(
        0,
        N_orbit_groups*orbit_group_size,
        orbit_group_size)
    # Now make empirical formula
    def E_of_years(x, *coeff):
        return sum(coeff[i]*x**float((len(coeff)-1)-i) \
                   for i in range(len(coeff)))
    E =  np.array(E)
    degree = 4
    # note index: polyfit finds p[0]*x**4 + p[1]*x**3 ...etc.
    p = np.polyfit(years, E, degree)
```

```
    p_str = map(str, p)
    formula = ' + '.join([p_str[i] + '*x**' + \
                      str(degree-i) for i in range(degree+1)])
    print 'Empirical formula (error with years):  ', formula
    plt.figure()
    plt.plot(years,
             E, 'b-',
             years,
             E_of_years(years, *p), 'r--')
    plt.xlabel('Number of years')
    plt.ylabel('Orbit error')
    plt.title(solver_ID)
    filename = solver_ID + 'tmp_E_with_years'
    plt.savefig(filename + '.png')
    plt.savefig(filename + '.pdf')
    plt.show()
    print 'Predicted CPU time in hours (1 billion years):', \
                                          cpu_time*10000
    print 'Predicted max error (1 billion years):', \
                                       E_of_years(1E9, *p)
compute_orbit_error_and_CPU():
'''
Orbit error and associated CPU times are computed with
solvers: RK4, Euler-Cromer, PEFRL.'''
def f_EC(u, t):
    '''
    Return derivatives for the 1st order system as
    required by Euler-Cromer.
    '''
    vx, x, vy, y = u  # u: array holding vx, x, vy, y
    d = -(x**2 + y**2)**(-3.0/2)
    return [d*x, vx, d*y, vy ]
def f_RK4(u, t):
    '''
    Return derivatives for the 1st order system as
    required by ordinary solvers in Odespy.
    '''
    x, vx, y, vy = u  # u: array holding x, vx, y, vy
    d = -(x**2 + y**2)**(-3.0/2)
    return [vx, d*x, vy, d*y ]
def g(u, v):
    '''
    Return derivatives for the 1st order system as
    required by PEFRL.
    '''
    d = -(u[0]**2 + u[1]**2)**(-3.0/2)
    return np.array([d*u[0], d*u[1]])
```

The standard way of solving the ODE by Odespy is then

```
def u_exact(t):
    """Return exact solution at time t."""
    return np.array([np.cos(t), np.sin(t)])

u_e = u_exact(time_points).transpose()

solver = odespy.RK4(f_RK4)
solver.set_initial_condition(A)
ui, ti = solver.solve(time_points)

# Find error (correct final pos:  x=1, y=0)
orbit_error = np.sqrt(
         (ui[:,0]-u_e[:,0])**2 + (ui[:,2]-u_e[:,1])**2).max()
```

We develop functions for computing errors and plotting results where we can compare different methods. These functions are shown in the solution to item d).

Running the code, the time step sizes become

```
dt_values: [0.031415926535897934, 0.015707963267948967,
            0.007853981633974483, 0.003926990816987242]
```

Corresponding maximum errors (per cent) and CPU values (hours) are for the 4th-order Runge-Kutta given in the table below.

Quantity	Δt_1	Δt_2	Δt_3	Δt_4
Δt	0.03	0.02	0.008	0.004
Error	1.9039	0.0787	0.0025	7.7e-05
CPU (h)	0.03	0.06	0.12	0.23

For Euler-Cromer we these results:

Quantity	Δt_1	Δt_2	Δt_3	Δt_4
Δt	0.03	0.02	0.008	0.004
Error	2.0162	2.0078	1.9634	0.6730
CPU (h)	0.01	0.02	0.05	0.09

These results are as expected. The Runge-Kutta implementation is much more accurate than Euler-Cromer, but since it requires more computations, more CPU time is needed. For both methods, accuracy and CPU time both increase as the step size is reduced, but the increase is much more pronounced for the Euler-Cromer method.

d) Implement a solver based on the PEFRL method from Section 10.11. Verify its 4th-order convergence using an equation $u'' + u = 0$.

Solution.

Here is a solver function:

```
import numpy as np
import time

def solver_PEFRL(I, V, g, dt, T):
    """
    Solve v' = - g(u,v), u'=v for t in (0,T], u(0)=I and v(0)=V,
    by the PEFRL method.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = np.zeros((Nt+1, len(I)))
    v = np.zeros((Nt+1, len(I)))
    t = np.linspace(0, Nt*dt, Nt+1)

    # these values are from eq (20), ref to paper below
    xi = 0.1786178958448091
    lambda_ = -0.2123418310626054
    chi = -0.06626458266981849

    v[0] = V
    u[0] = I
    # Compare with eq 22 in http://arxiv.org/pdf/cond-mat/0110585.pdf
    for n in range(0, Nt):
        u_ = u[n] + xi*dt*v[n]
        v_ = v[n] + 0.5*(1-2*lambda_)*dt*g(u_, v[n])
        u_ = u_ + chi*dt*v_
        v_ = v_ + lambda_*dt*g(u_, v_)
        u_ = u_ + (1-2*(chi+xi))*dt*v_
        v_ = v_ + lambda_*dt*g(u_, v_)
        u_ = u_ + chi*dt*v_
        v[n+1] = v_ + 0.5*(1-2*lambda_)*dt*g(u_, v_)
        u[n+1] = u_ + xi*dt*v[n+1]
        #print 'v[%d]=%g, u[%d]=%g' % (n+1,v[n+1],n+1,u[n+1])
    return u, v, t
```

A proper test function for verification reads

```
def test_solver_PEFRL():
    """Check 4th order convergence rate, using u'' + u = 0,
    I = 3.0, V = 0, which has the exact solution u_e = 3*cos(t)"""
    def g(u, v):
        return np.array([-u])
    def u_exact(t):
        return np.array([3*np.cos(t)]).transpose()
    I = u_exact(0)
```

```
V = np.array([0])
print 'V:', V, 'I:', I

# Numerical parameters
w = 1
P = 2*np.pi/w
dt_values = [P/20, P/40, P/80, P/160, P/320]
T = 8*P
error_vs_dt = []
for n, dt in enumerate(dt_values):
    u, v, t = solver_PEFRL(I, V, g, dt, T)
    error = np.abs(u - u_exact(t)).max()
    print 'error:', error
    if n > 0:
        error_vs_dt.append(error/dt**4)
for i in range(1, len(error_vs_dt)):
    #print abs(error_vs_dt[i]- error_vs_dt[0])
    assert abs(error_vs_dt[i]-
               error_vs_dt[0]) < 0.1
```

e) The simulations done previously with the 4th-order Runge-Kutta and Euler-Cromer are now to be repeated with the PEFRL solver, so the code must be extended accordingly. Then run the simulations and comment on the performance of PEFRL compared to the other two.

Solution.

With the PEFRL algorithm, we get

```
E max with dt...: [0.0010452575786173163, 6.5310955829464402e-05,
                   4.0475768394248492e-06, 2.9391302503251016e-07]
cpu_values with dt...: [0.01873611111111106, 0.037422222222222294,
                        0.07511666666666655, 0.14985]
```

Qantity	Δt_1	Δt_2	Δt_3	Δt_4
Δt	0.03	0.02	0.008	0.004
Error	1.04E-3	6.53E-05	4.05E-6	2.94E-7
CPU (h)	0.02	0.04	0.08	0.15

The accuracy is now dramatically improved compared to 4th-order Runge-Kutta (and Euler-Cromer). With 1600 steps per orbit, the PEFRL maximum error is just below $3.0e-07$ per cent, while the corresponding error with Runge-Kutta was about $7.7e-05$ per cent! This is striking, considering the fact that the 4th-order Runge-Kutta and the PEFRL schemes are both 4th-order accurate.

f) Use the PEFRL solver to simulate 100000 orbits with a fixed time step corresponding to 1600 steps per period. Record the maximum error within each subsequent group of 1000 orbits. Plot these errors and fit (least squares) a mathematical function to the data. Print also the total CPU time spent for all 100000 orbits.

Now, predict the error and required CPU time for a simulation of 1 billion years (orbits). Is it feasible on today's computers to simulate the planetary motion for one billion years?

Solution.

The complete code (which also produces the printouts given previously) reads:

```
import scitools.std as plt
import sys
import odespy
import numpy as np
import time

def solver_PEFRL(I, V, g, dt, T):
    """
    Solve v' = - g(u,v), u'=v for t in (0,T], u(0)=I and v(0)=V,
    by the PEFRL method.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = np.zeros((Nt+1, len(I)))
    v = np.zeros((Nt+1, len(I)))
    t = np.linspace(0, Nt*dt, Nt+1)

    # these values are from eq (20), ref to paper below
    xi = 0.1786178958448091
    lambda_ = -0.2123418310626054
    chi = -0.06626458266981849

    v[0] = V
    u[0] = I
    # Compare with eq 22 in http://arxiv.org/pdf/cond-mat/0110585.pdf
    for n in range(0, Nt):
        u_ = u[n] + xi*dt*v[n]
        v_ = v[n] + 0.5*(1-2*lambda_)*dt*g(u_, v[n])
        u_ = u_ + chi*dt*v_
        v_ = v_ + lambda_*dt*g(u_, v_)
        u_ = u_ + (1-2*(chi+xi))*dt*v_
        v_ = v_ + lambda_*dt*g(u_, v_)
        u_ = u_ + chi*dt*v_
        v[n+1] = v_ + 0.5*(1-2*lambda_)*dt*g(u_, v_)
```

```
        u[n+1] = u_ + xi*dt*v[n+1]
        #print 'v[%d]=%g, u[%d]=%g' % (n+1,v[n+1],n+1,u[n+1])
    return u, v, t

def test_solver_PEFRL():
    """Check 4th order convergence rate, using u'' + u = 0,
    I = 3.0, V = 0, which has the exact solution u_e = 3*cos(t)"""
    def g(u, v):
        return np.array([-u])
    def u_exact(t):
        return np.array([3*np.cos(t)]).transpose()
    I = u_exact(0)
    V = np.array([0])
    print 'V:', V, 'I:', I

    # Numerical parameters
    w = 1
    P = 2*np.pi/w
    dt_values = [P/20, P/40, P/80, P/160, P/320]
    T = 8*P
    error_vs_dt = []
    for n, dt in enumerate(dt_values):
        u, v, t = solver_PEFRL(I, V, g, dt, T)
        error = np.abs(u - u_exact(t)).max()
        print 'error:', error
        if n > 0:
            error_vs_dt.append(error/dt**4)
    for i in range(1, len(error_vs_dt)):
        #print abs(error_vs_dt[i]- error_vs_dt[0])
        assert abs(error_vs_dt[i]-
                   error_vs_dt[0]) < 0.1


class PEFRL(odespy.Solver):
    """Class wrapper for Odespy."""  # Not used!
    quick_desctiption = "Explicit 4th-order method for v'=-f, u=v."""

    def advance(self):
        u, f, n, t = self.u, self.f, self.n, self.t
        dt = t[n+1] - t[n]
        I = np.array([u[1], u[3]])
        V = np.array([u[0], u[2]])
        u, v, t = solver_PFFRL(I, V, f, dt, t+dt)
        return np.array([v[-1], u[-1]])

def compute_orbit_and_error(
    f,
    solver_ID,
```

```
timesteps_per_period=20,
N_orbit_groups=1000,
orbit_group_size=10):
'''
For one particular solver:
Calculte the orbits for a multiple of grouped orbits, i.e.
number of orbits = orbit_group_size*N_orbit_groups.
Returns: time step dt, and, for each N_orbit_groups cycle,
the 2D position error and cpu time (as lists).
'''
def u_exact(t):
    return np.array([np.cos(t), np.sin(t)])

w = 1
P = 2*np.pi/w       # scaled period (1 year becomes 2*pi)
dt = P/timesteps_per_period
Nt = orbit_group_size*N_orbit_groups*timesteps_per_period
T = Nt*dt
t_mesh = np.linspace(0, T, Nt+1)
E_orbit = []

#print '        dt:', dt
T_interval = P*orbit_group_size
N = int(round(T_interval/dt))

# set initial conditions
if solver_ID == 'EC':
    A = [0,1,1,0]
elif solver_ID == 'PEFRL':
    I = np.array([1, 0])
    V = np.array([0, 1])
else:
    A = [1,0,0,1]

t1 = time.clock()
for i in range(N_orbit_groups):
    time_points = np.linspace(i*T_interval, (i+1)*T_interval,N+1)
    u_e = u_exact(time_points).transpose()
    if solver_ID == 'EC':
        solver = odespy.EulerCromer(f)
        solver.set_initial_condition(A)
        ui, ti = solver.solve(time_points)
        # Find error (correct final pos:  x=1, y=0)
        orbit_error = np.sqrt(
            (ui[:,1]-u_e[:,0])**2 + (ui[:,3]-u_e[:,1])**2).max()
    elif solver_ID == 'PEFRL':
        # Note: every T_inverval is here counted from time 0
        ui, vi, ti = solver_PEFRL(I, V, f, dt, T_interval)
```

```
            # Find error (correct final pos:  x=1, y=0)
            orbit_error = np.sqrt(
                (ui[:,0]-u_e[:,0])**2 + (ui[:,1]-u_e[:,1])**2).max()
        else:
            solver = eval('odespy.' + solver_ID(f)
            solver.set_initial_condition(A)
            ui, ti = solver.solve(time_points)
            # Find error (correct final pos:  x=1, y=0)
            orbit_error = np.sqrt(
                (ui[:,0]-u_e[:,0])**2 + (ui[:,2]-u_e[:,1])**2).max()

        print '      Orbit no. %d,   max error (per cent): %g' % \
                      ((i+1)*orbit_group_size, orbit_error)

        E_orbit.append(orbit_error)

        # set init. cond. for next time interval
        if solver_ID == 'EC':
            A = [ui[-1,0], ui[-1,1], ui[-1,2], ui[-1,3]]
        elif solver_ID == 'PEFRL':
            I = [ui[-1,0], ui[-1,1]]
            V = [vi[-1,0], vi[-1,1]]
        else:  # RK4, adaptive rules, etc.
            A = [ui[-1,0], ui[-1,1], ui[-1,2], ui[-1,3]]

    t2 = time.clock()
    CPU_time = (t2 - t1)/(60.0*60.0)    # in hours
    return dt, E_orbit, CPU_time

def orbit_error_vs_dt(
    f_EC, f_RK4, g, solvers,
    N_orbit_groups=1000,
    orbit_group_size=10):
    '''
    With each solver in list "solvers": Simulate
    orbit_group_size*N_orbit_groups orbits with different dt values.
    Collect final 2D position error for each dt and plot all errors.
    '''

    for solver_ID in solvers:
        print 'Computing orbit with solver:', solver_ID
        E_values = []
        dt_values = []
        cpu_values = []
        for timesteps_per_period in 200, 400, 800, 1600:
            print '.......time steps per period: ', \
                                        timesteps_per_period
            if solver_ID == 'EC':
```

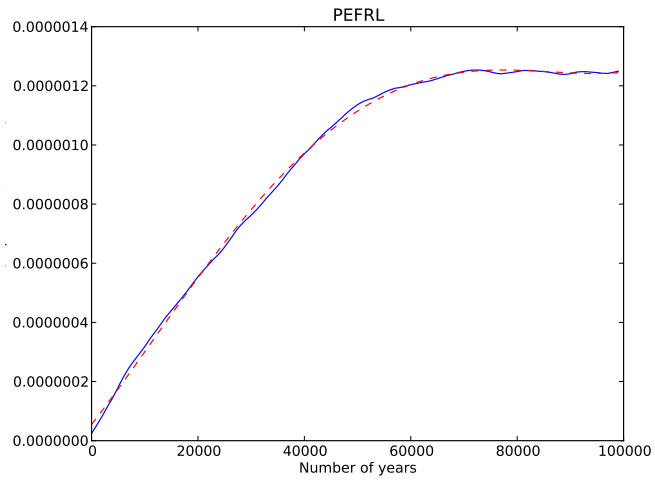
```
                dt, E, cpu_time = compute_orbit_and_error(
                    f_EC,
                    solver_ID,
                    timesteps_per_period,
                    N_orbit_groups,
                    orbit_group_size)
            elif solver_ID == 'PEFRL':
                dt, E, cpu_time = compute_orbit_and_error(
                    g,
                    solver_ID,
                    timesteps_per_period,
                    N_orbit_groups,
                    orbit_group_size)
            else:
                dt, E, cpu_time = compute_orbit_and_error(
                    f_RK4,
                    solver_ID,
                    timesteps_per_period,
                    N_orbit_groups,
                    orbit_group_size)

            dt_values.append(dt)
            E_values.append(np.array(E).max())
            cpu_values.append(cpu_time)
        print 'dt_values:', dt_values
        print 'E max with dt...:', E_values
        print 'cpu_values with dt...:', cpu_values


def orbit_error_vs_years(
    f_EC, f_RK4, g, solvers,
    N_orbit_groups=1000,
    orbit_group_size=100,
    N_time_steps = 1000):
    '''
    For each solver in the list solvers:
    simulate orbit_group_size*N_orbit_groups orbits with a fixed
    dt corresponding to N_time_steps steps per year.
    Collect max 2D position errors for each N_time_steps'th run,
    plot these errors and CPU. Finally, make an empirical
    formula for error and CPU as functions of a number
    of cycles.
    '''
    timesteps_per_period = N_time_steps     # fixed for all runs

    for solver_ID in solvers:
        print 'Computing orbit with solver:', solver_ID
        if solver_ID == 'EC':
```

```
    dt, E, cpu_time = compute_orbit_and_error(
        f_EC,
        solver_ID,
        timesteps_per_period,
        N_orbit_groups,
        orbit_group_size)
elif solver_ID == 'PEFRL':
    dt, E, cpu_time = compute_orbit_and_error(
        g,
        solver_ID,
        timesteps_per_period,
        N_orbit_groups,
        orbit_group_size)
else:
    dt, E, cpu_time = compute_orbit_and_error(
        f_RK4,
        solver_ID,
        timesteps_per_period,
        N_orbit_groups,
        orbit_group_size)

# E and cpu_time are for every N_orbit_groups cycle
print 'E_values (fixed dt, changing no of years):', E
print 'CPU (hours):', cpu_time
years = np.arange(
    0,
    N_orbit_groups*orbit_group_size,
    orbit_group_size)

# Now make empirical formula

def E_of_years(x, *coeff):
    return sum(coeff[i]*x**float((len(coeff)-1)-i) \
               for i in range(len(coeff)))
E =  np.array(E)
degree = 4
# note index: polyfit finds p[0]*x**4 + p[1]*x**3 ...etc.
p = np.polyfit(years, E, degree)
p_str = map(str, p)
formula = ' + '.join([p_str[i] + '*x**' + \
                str(degree-i) for i in range(degree+1)])

print 'Empirical formula (error with years):  ', formula
plt.figure()
plt.plot(years,
         E, 'b-',
         years,
         E_of_years(years, *p), 'r--')
```

```
        plt.xlabel('Number of years')
        plt.ylabel('Orbit error')
        plt.title(solver_ID)
        filename = solver_ID + 'tmp_E_with_years'
        plt.savefig(filename + '.png')
        plt.savefig(filename + '.pdf')
        plt.show()

        print 'Predicted CPU time in hours (1 billion years):', \
                                                cpu_time*10000
        print 'Predicted max error (1 billion years):', \
                                          E_of_years(1E9, *p)

def compute_orbit_error_and_CPU():
    '''
    Orbit error and associated CPU times are computed with
    solvers: RK4, Euler-Cromer, PEFRL.'''

    def f_EC(u, t):
        '''
        Return derivatives for the 1st order system as
        required by Euler-Cromer.
        '''
        vx, x, vy, y = u  # u: array holding vx, x, vy, y
        d = -(x**2 + y**2)**(-3.0/2)
        return [d*x, vx, d*y, vy ]

    def f_RK4(u, t):
        '''
        Return derivatives for the 1st order system as
        required by ordinary solvers in Odespy.
        '''
        x, vx, y, vy = u  # u: array holding x, vx, y, vy
        d = -(x**2 + y**2)**(-3.0/2)
        return [vx, d*x, vy, d*y ]

    def g(u, v):
        '''
        Return derivatives for the 1st order system as
        required by PEFRL.
        '''
        d = -(u[0]**2 + u[1]**2)**(-3.0/2)
        return np.array([d*u[0], d*u[1]])

    print 'Find orbit error as fu. of dt...(10000 orbits)'
    solvers = ['RK4', 'EC', 'PEFRL']
    N_orbit_groups=1
    orbit_group_size=10000
```

```
    orbit_error_vs_dt(
        f_EC, f_RK4, g, solvers,
        N_orbit_groups=N_orbit_groups,
        orbit_group_size=orbit_group_size)

    print 'Compute orbit error as fu. of no of years (fixed dt)...'
    solvers = ['PEFRL']
    N_orbit_groups=100
    orbit_group_size=1000
    N_time_steps = 1600    # no of steps per orbit cycle
    orbit_error_vs_years(
        f_EC, f_RK4, g, solvers,
        N_orbit_groups=N_orbit_groups,
        orbit_group_size=orbit_group_size,
        N_time_steps = N_time_steps)

if __name__ == '__main__':
    test_solver_PEFRL()
    compute_orbit_error_and_CPU()
```

The maximum error develops with number of orbits as seen in the following plot, where the red dashed curve is from the mathematical model:



We note that the maximum error achieved during the first 100000 orbits is only about $1.2e-06$ per cent. Not bad!

For the printed CPU and empirical formula, we get:

```
CPU (hours): 1.51591388889
Empirical formula (E with years):
3.15992325978e-26*x**4 + -6.1772567063e-21*x**3 +
1.87983349496e-16*x**2 + 2.32924158693e-11*x**1 +
5.46989368301e-08*x**0
```

Since the CPU develops linearly, the CPU time for 100000 orbits can just be multiplied by 10000 to get the estimated CPU time required for 1 billion years. This gives 15159 CPU hours (631 days), which is also printed.

With the derived empirical formula, the estimated orbit error after 1 billion years becomes 31593055529 per cent.

Filename: `vib_PEFRL`.

Remarks. This exercise investigates whether it is feasible to predict planetary motion for the life time of a solar system.

References

[1] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I. Nonstiff Problems*. Springer, 1993.

[2] H. P. Langtangen. *Finite Difference Computing with Exponential Decay Models*. Lecture Notes in Computational Science and Engineering. Springer, 2016. http://hplgit.github.io/decay-book/doc/web/.

[3] H. P. Langtangen and G. K. Pedersen. *Scaling of Differential Equations*. Simula Springer Brief Series. Springer, 2016. http://hplgit.github.io/scaling-book/doc/web/.

[4] I. P. Omelyan, I. M. Mryglod, and R. Folk. Optimized forest-ruth- and suzuki-like algorithms for integration of motion in many-body systems. *Computer Physics Communication*, 146(2):188–202, 2002.

Index