A Vibration Problem

Hans Petter Langtangen[1,2]

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

Sep 13, 2012

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

Contents

List of exercises

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

1 A simple vibration problem

Our first model problem for vibrations takes the form

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

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

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

That is, u oscillates with constant amplitude I and (angular) frequency ω. The corresponding period of oscillations (e.g., 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. The latter is not measured in Hz, but in radians per second.

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

1.1 A centered finite difference scheme

To formulate a finite difference method for the model problem (1) we follow the four steps from Section 1.2 in [**?**].

The discretization. First, the domain is discretized by introducing a uniformly partitioned time mesh in the present problem. The points in the mesh are hence $t_n = n\Delta t$, $n = 0, 1, \ldots, N$, where $\Delta t = T/N$ is the constant length of the time steps. Second, the ODE is to be satisfied at each mesh point:

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

Third, 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

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

Fourth, to formulate the computational algorithm, we assume that we have already computed u^{n-1} and u^n such that u^{n-1} is the unknown value, which we can readily solve for:

$$u^{n+1} = 2u^n - u^{n-1} - \omega^2 u^n \,. \tag{6}$$

The computational algorithm is simply to apply (6) successively for $n = 1, 2, \ldots, N-1$. This numerical scheme sometimes goes under the name Störmer's method or Verlet integration.

Computing the first step. We observe that (6) cannot be used for $n = 0$ since the computation of u^1 then involves the undefined value u^{-1} at $t = -\Delta t$. Also, we have not used the "other" initial condition $u'(0) = 0$. This condition can be combined with (6) for $n = 1$ to yield a value for u^1. We may use central difference approximation to $u'(0)$ of the type

$$\frac{u^1 - u^{-1}}{2\Delta t} = 0 \quad \Rightarrow \quad u^{-1} = u^1 \,. \tag{7}$$

Using this relation in (6) for $n = 0$, we get

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

which reduces to

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

Exercise 3 asks you to perform an alternative derivation and also to generalize the initial condition to $u'(0) = V \neq 0$.

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

1. $u^0 = I$
2. compute u^1 from (8)
3. for $n = 1, 2, \ldots, N-1$:
   (a) compute u^{n+1} from (6)

The algorithm is more precisely expressed directly in Python:

```
t = linspace(0, T, N+1)  # mesh points in time
dt = t[1] - t[0]         # constant time step.
u = zeros(N+1)           # solution

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

We remark that `w` is used as symbol in the code for ω. The reason is that this author prefers `w` for readability and comparison with the mathematical ω instead of the full word `omega` as variable name.

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

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

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

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

which is written out as

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

The discretization of initial conditions can in the operator notation be expressed as

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

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

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

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

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

1.2 Implementation

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

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

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

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

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

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

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

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

```
I = 1
w = 2*pi
dt = 0.05
num_periods = 5
P = 2*pi/w    #  one period
T = P*num_periods
u, t = solver(I, w, dt, T)
visualize(u, t, I, w, dt)
```

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

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--I', type=float, default=1.0)
parser.add_argument('--w', type=float, default=2*pi)
parser.add_argument('--dt', type=float, default=0.05)
parser.add_argument('--num_periods', type=int, default=5)
a = parser.parse_args()
I, w, dt, num_periods = a.I, a.w, a.dt, a.num_periods
```

1.3 Verification

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

Testing very simple solutions. Constructing test problems where the exact solution is constant or linear helps initial debugging and verification as one expects any reasonable numerical method to reproduce such solutions to machine precision. We would, however, need a source term $f(t)$ in the equation, $u'' + \omega^2 u = f(t)$, to have a constant solution more complicated than the not so useful $u = 0$.

Checking convergence rates. Empirical computation of convergence rates, as explained in Section 2.7 in [**?**], yields a good method for verification. The function below

- performs m simulations with halved time steps: $2^{-k}\Delta t$, $k = 0, \ldots, m-1$,
- computes the L_2 norm of the error, $E = \sqrt{\Delta t_i \sum_{n=0}^{N-1}(u^n - u_{\rm e}(t_n))^2}$ in each case,
- estimates the rates r_i from two consecutive experiments $(\Delta t_{i-1}, E_{i-1})$ and $(\Delta t_i, E_i)$, assuming $E_i = C\Delta t_i^{r_i}$ and $E_{i-1} = C\Delta t_{i-1}^{r_i}$:

The implementational details goes as follows:

```
def convergence_rates(m, num_periods=8):
    """
    Return m-1 empirical estimates of the convergence rate
    based on m simulations, where the time step is halved
    for each simulation.
    """
    w = 0.35; I = 0.3
    dt = 2*pi/w/30  # 30 time step per period 2*pi/w
    T = 2*pi/w*num_periods
    dt_values = []
    E_values = []
    for i in range(m):
        u, t = solver(I, w, dt, T)
        u_e = exact_solution(t, I, w)
        E = sqrt(dt*sum((u_e-u)**2))
        dt_values.append(dt)
        E_values.append(E)
        dt = dt/2

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

The return `r` list has its values equal to 2.00, which is in excellent agreement with what is expected from the second-order finite difference approximation $[D_tD_tu]^n$. The final `r[-1]` value is a good candidate for a unit test:

```
def test_convergence_rates():
    r = convergence_rates(m=5, num_periods=8)
    # Accept rate to 1 decimal place
    nt.assert_almost_equal(r[-1], 2.0, places=1)
```

The complete code appears in `vb_undamped.py`.

1.4 Long-time simulations

Figure 1 shows a comparison of the exact and numerical solution for $\Delta t = 0.1, 0.05$ and $w = 2\pi$. From the plot we make the following observations:

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

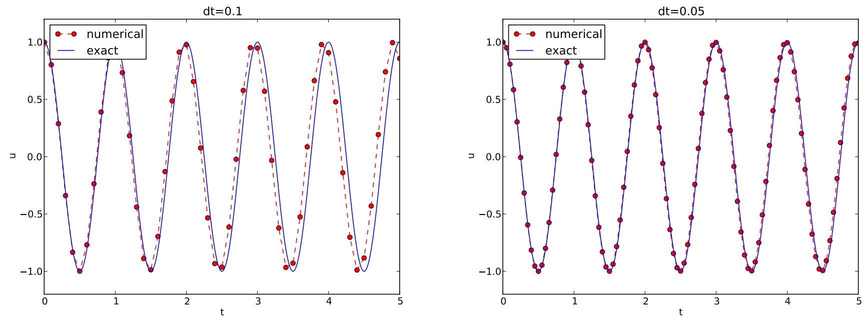

Figure 1: Effect of halving the time step.

Using a moving plot window. In vibration problems it is often of interest to investigate the system's behavior over long time intervals. Errors in the phase may then show up as crucial. Let us investigate long time series by introducing a moving plot window that can move along with the p most recently computed periods of the solution. The SciTools package contains a convenient tool for this: `MovingPlotWindow`. Typing `pydoc scitools.MovingPlotWindow` shows a demo and description of usage. The function below illustrates the usage and is invoked in the `vb_undamped.py` code if the number of periods in the simulation exceeds 10:

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

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

Running

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

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

The `visualize_font` function stores all the plots in files with names `tmp_vib0000.png`, `tmp_vib0001.png`, `tmp_vib0002.png`, and so on. From these files we may make a movie. This is particularly easy and convenient with the `scitools movie` command. The simplest movie format is a web page where the PNG files can be displayed consecutively. The generation of such a web page goes like

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

The `fps` argument controls the speed of the movie ("frames per second"). To ensure that the individual plot frames are shown in correct order, it is important to number the files with zero-padded numbers (0000, 0001, 0002, etc.). The printf format `%04d` specifies an integer in a field of width 4, padded with zeros from the left. A simple Unix wildcard file specification like `tmp_vib*.png` will then list the frames in the right order. (If the numbers in the filenames were not zero-padded, `tmp_vib11.png` would appear before `tmp_vib2.png`!)

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

```
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_vib*.png`:

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

The `scitools movie` command can make movies in different formats, depending upon what software that is installed on the computer. If you have the `convert` program from the ImageMagick suite, an animated GIF file can be made by

```
Terminal> scitools movie encoder=convert output_file=vib.gif \
          fps=4 tmp_vib*.png
```

One can alternatively use `convert` directly:

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

However, 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 feasible.

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

```
def visualize_front_ascii(u, t, I, w, fps=10):
    """
    Plot u and the exact solution vs t line by line in a
    terminal window (only using ascii characters).
    Makes it easy to plot very long time series.
    """
    from scitools.avplotter import Plotter
    import time
    P = 2*pi/w
    umin = -1.2*I;  umax = -umin

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

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

```
                         |                       o+      14.0
                         |                      + o      14.0
                         |                  +    o       14.1
                         |             +     o           14.1
                         |     +        o                14.2
                        +|       o                       14.2
                +        |                               14.2
         +       o       |                               14.3
    +     o              |                               14.4
 +   o                   |                               14.4
+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
```

1.5 Analysis of the numerical scheme

Deriving an exact numerical solution. After having seen the phase error grow with time in the previous section, we shall now quantify this error through mathematical analysis. The key tool in the analysis will be to establish an exact solution of the discrete equations. The difference equation (6) has constant coefficients and is homogeneous. The solution is then of the form $u^n = A^n$, where A is some number to be determined. Since we have oscillating functions as solutions, the algebra will be considerably simplified if we write $A = \exp{(i\tilde{\omega}\Delta t)}$, which means

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

The physically relevant numerical solution can be taken as the real part of this complex expression. With the rewrite $A = \exp{(i\tilde{\omega}\Delta t)}$, the numerical frequency $\tilde{\omega}$ is the quantity to determine.

Calculations now give

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

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 (6) with $u^n = \exp{(i\omega\tilde{\Delta} t\, n)}$ inserted gives

$$-\exp{(i\tilde{\omega} t)}\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2}) + \omega^2\exp{(i\tilde{\omega} t)} = 0, \tag{13}$$

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

$$\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2}) = \omega^2\,. \tag{14}$$

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

$$\tilde{\omega} = \pm\frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right)\,. \tag{15}$$

The first observation of (15) 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 (15)? Taylor series expansion for small Δt may give an expression that is easier to understand:

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

This means that

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

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

Figure 2 plots the discrete frequency (15) and its approximation (16) for $w = 1$, based on the program `vb_plot_freq.py`). Although $\tilde{\omega}$ is a function of Δt in ref(16), it is instructive to replace Δt by the number of time steps per period in the solution, because $\omega\Delta t$ is the key parameter in the problem and this parameter reflects how many time steps we have per period. The plot shows that at least 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 1).

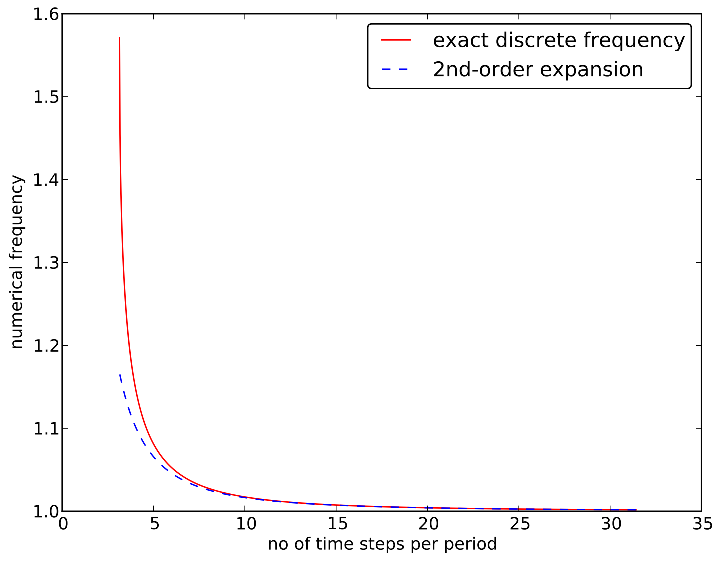

Figure 2: Exact discrete frequency and its second-order series expansion.

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

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

Such an exact discrete solution is ideal for verification purposes (and you are encouraged to make a test based on (17) in Exercise **??**).

Stability. Looking at (17), it appears that the numerical solution has constant and correct amplitude, but an error in the frequency (phase error). However, a constant amplitude demands that $\tilde{\omega}$ is a real number. A complex $\tilde{\omega}$ is indeed possible if the argument x of $\sin^{-1}(x)$ has magnitude larger than unity: $|x| > 1$ (type `asin(x)` in wolframalpha.com 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 $x > 1$, it means that $\exp\left(i\omega\tilde{t}\right) = \exp\left(-\tilde{\omega}_i t\right)\exp\left(i\tilde{\omega}_r t\right)$ will lead to exponential growth in time because $\tilde{\omega}_i < 0$ and hence $-\tilde{\omega}_i t > 0$.

We do not tolerate growth in the amplitude and we therefore have a *stability criterion*

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

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

From the analysis we can draw three important conclusions:

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

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

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

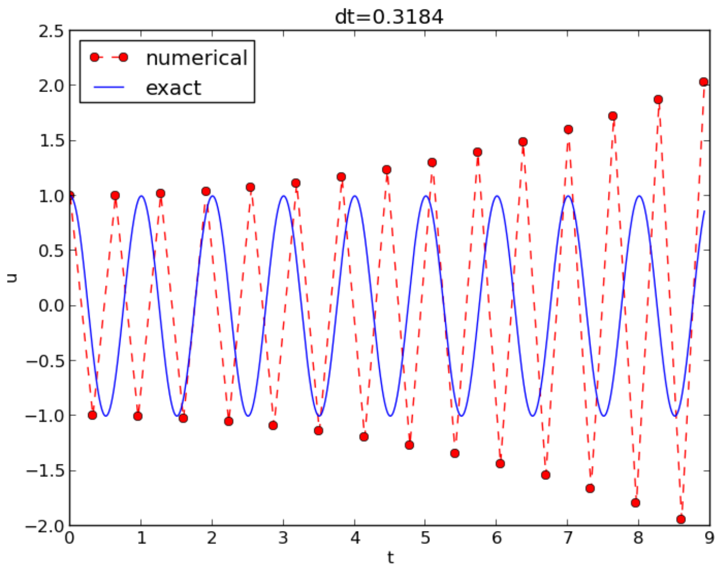

Figure 3: Growing, unstable solution because of a time step slightly beyond the stability limit.

2 Alternative schemes based on 1st-order equations

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

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

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

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

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

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

2.1 Standard methods for 1st-order ODE systems

The Forward Euler scheme. A Forward Euler approximation to our 2×2 system of ODEs (19)-(20) becomes

$$[D_t^+ u = v]^n, [D_t^+ v = -\omega^2 u]^n, \tag{21}$$

or written out,

$$u^{n+1} = u^n + \Delta t v^n, \tag{22}$$
$$v^{n+1} = v^n - \Delta t \omega^2 u^n\,. \tag{23}$$

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

$$[D_t^- u = v]^{n+1}, \tag{24}$$
$$[D_t^- v = -\omega u]^{n+1}\,. \tag{25}$$

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

$$u^{n+1} - \Delta t v^{n+1} = u^n, \tag{26}$$
$$v^{n+1} + \Delta t \omega^2 u^{n+1} = v^n\,. \tag{27}$$

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

$$[D_t u = \overline{v}^t]^{n+\frac{1}{2}}, \tag{28}$$
$$[D_t v = -\omega \overline{u}^t]^{n+\frac{1}{2}}\,. \tag{29}$$

Also a coupled system:

$$u^{n+1} - \frac{1}{2}\Delta t v^{n+1} = u^n + \frac{1}{2}\Delta t v^n, \tag{30}$$
$$v^{n+1} + \frac{1}{2}\Delta t \omega^2 u^{n+1} = v^n - \frac{1}{2}\Delta t \omega^2 u^n\,. \tag{31}$$

Comparison of schemes. We can easily compare methods like the ones above with the aid of the Odespy package:

```
import odespy
import numpy as np

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

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

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

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

Specification of the Forward Euler, Backward Euler, and Crank-Nicolson schemes is done like this (the equivalent to Crank-Nicolson in Odespy is `MidpointImplicit`):

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

The `vb_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 because the axes are automatically scaled. The important feature, however, is that exact curve $(u(t), v(t))$ is closed and repeats itself for every period. Not all numerical schemes are capable to do that.

The Forward Euler scheme in Figure 4 has a pronounced spiral curve, pointing to the fact that the amplitude steadily grows, which is also evident in Figure 5. The Backward Euler scheme has a similar feature, except that the spriral goes inward and the amplitude is significantly damped. The changing amplitude and the sprial form decreases with decreasing time step. The Crank-Nicolson scheme (MidpointImplicit) 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.

We may run two popular standard methods for first-order ODEs, the 2nd- and 4th-order Runge-Kutta methods, to see how they perform. Figures 6 and 7 show the solutions with larger Δt values than what was used in the previous two plots. How the amplitude develops in longer time integrations is illustrated in Figures 8 and 9. The markers are dropped in these plots because there are so many mesh points with markers when T corresponds to 10 periods. The visual impression is that the 4th-order Runge-Kutta method is very accurate, under

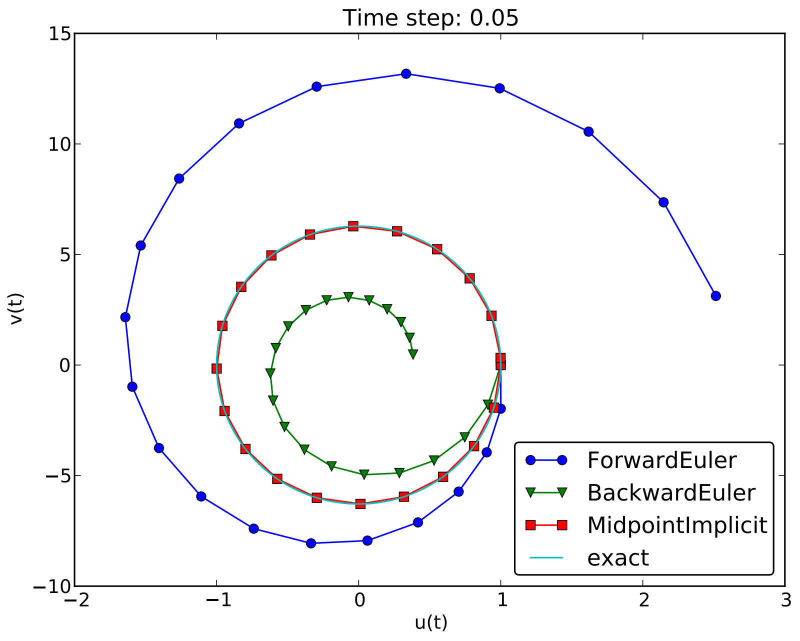

Figure 4: Comparison of classical schemes in the phase plane.

all circumstances in these tests, and the 2nd-order scheme suffer from amplitude errors unless the time step is very small.

The corresponding results for the Crank-Nicolson scheme are shown in Figures 10 and 11. It is clear that scheme outperforms the 2nd-order Runge-Kutta method. Both schemes has the same order of accuracy, but their differences in accuracy is clearly pronounced in this example.

2.2 The Euler-Cromer method

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

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

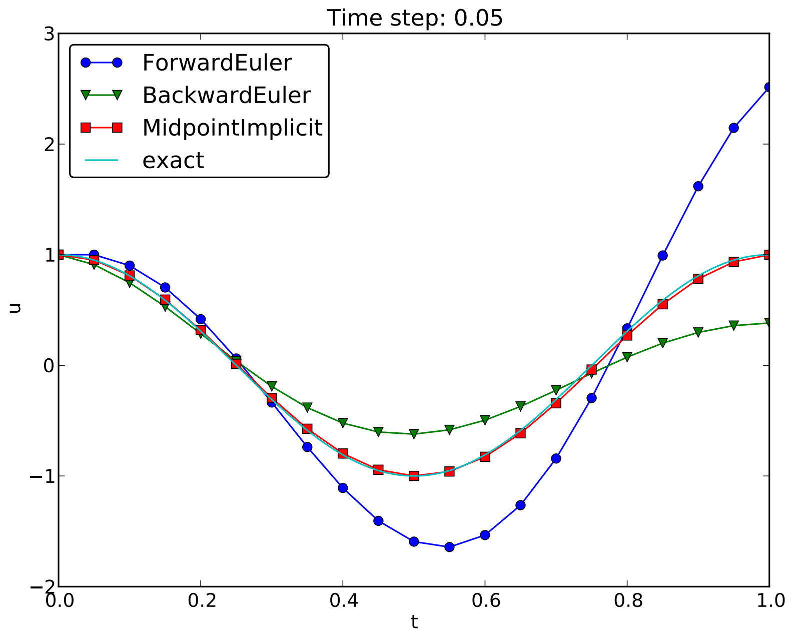

Figure 5: Comparison of classical schemes.

$$[D_t^+ u = v]^n, \tag{32}$$

$$[D_t^- v = -\omega u]^{n+1} . \tag{33}$$

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

$$u^{n+1} = u^n + \Delta t v^n, \tag{34}$$

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

We realize that u^{n+1} can be computed from (34) and then v^{n+1} from (35) using the recently computed value u^{n+1} on the right-hand side.

The scheme (34)-(35) goes under several names: Forward-backward scheme, Semi-implicit Euler method, symplectic Euler, semi-explicit Euler, Newton-Stormer-Verlet, and Euler-Cromer. Since both discretizations are based on first-order difference approximation, one may think that the scheme is only of first-order, but this is not true: the use of a forward and then a backward difference make errors cancel so that the overall error in the scheme os $\mathcal{O}(\Delta t^2)$. This is explaned below.

Equivalence with the scheme for the second-order ODE. We may eliminate the v^n variable from (34)-(35). From (35) we have $v^n = v^{n-1} - \Delta t\omega^2 u^n$,

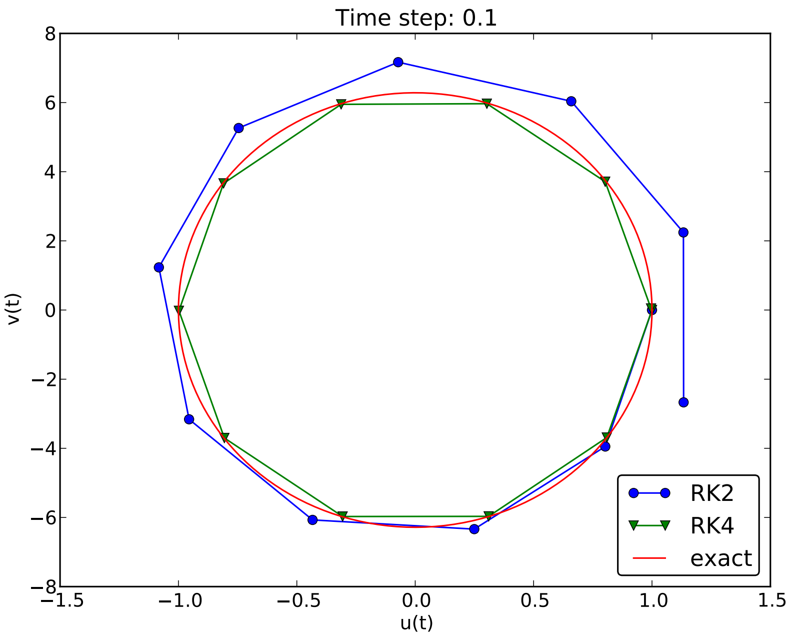

Figure 6: Comparison of Runge-Kutta schemes in the phase plane.

which can be inserted in (34) to yield

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

The v^{n-1} quantity can be expressed by u^n and u^{n-1} using (34):

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

and when this is inserted in (36) we get

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

which is nothing but the centered scheme (6)! The previous analysis of this scheme then also applies to the Euler-Cromer method.

The initial condition $u' = 0$ means $u' = v = 0$. Then $v^0 = 0$, and (34) implies $u^1 = u^0$, while (35) says $v^1 = -\omega^2 u^0$. This approximation, $u^1 = u^0$, corresponds to a first-order Forward Euler discretization of the initial condition $u'(0) = 0$: $[D_t^+ u = 0]^0$.

2.3 A method utilizing a staggered mesh

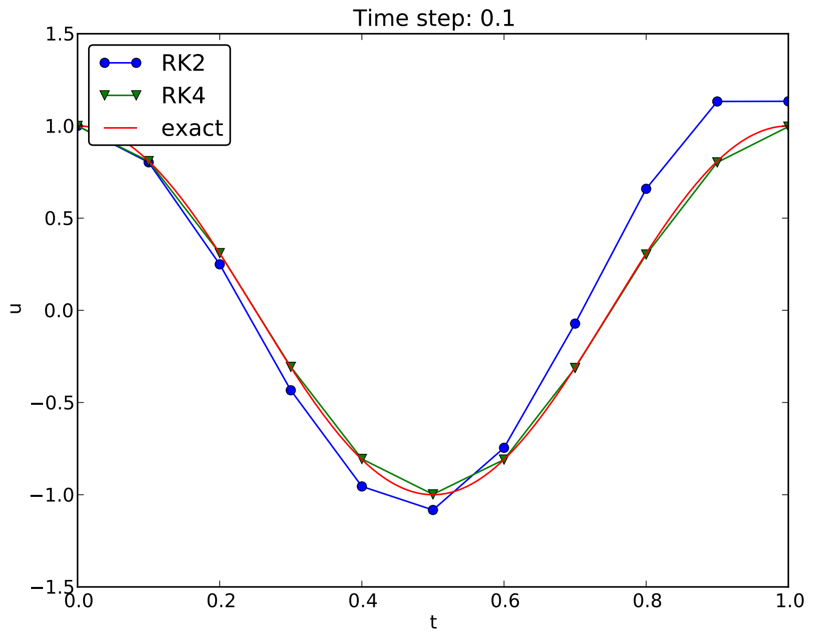

Figure 7: Comparison of Runge-Kutta schemes.

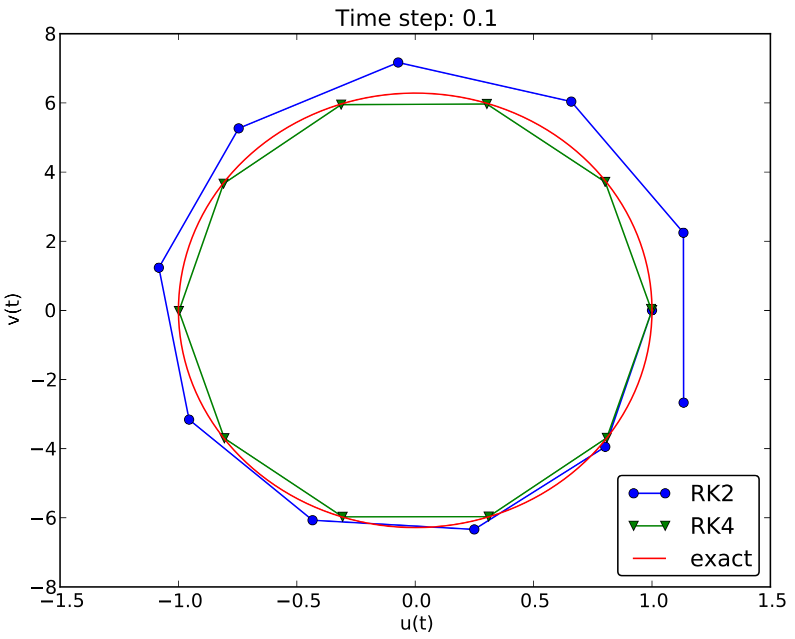

Figure 8: Long-time behavior of Runge-Kutta schemes in the phase plane.

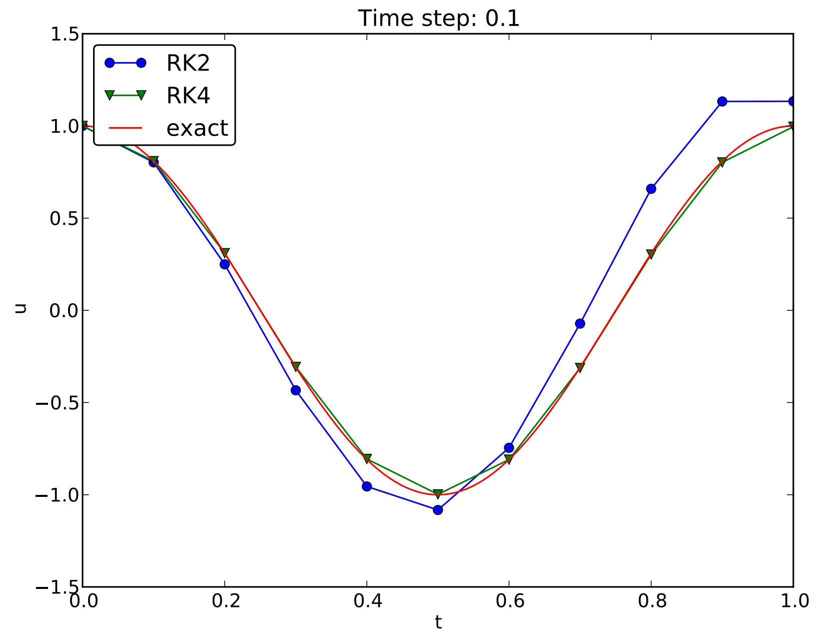

Figure 9: Long-time behavior of Runge-Kutta schemes.

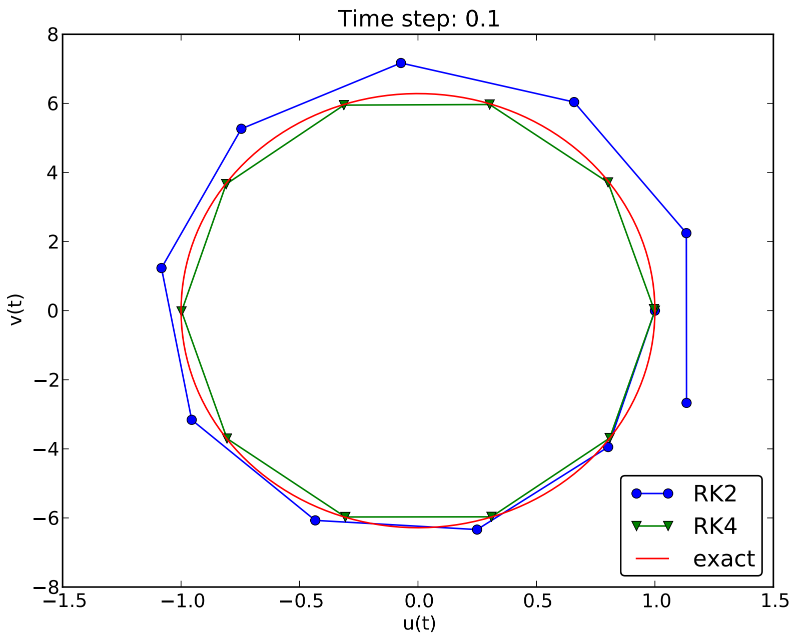

Figure 10: Long-time behavior of the Crank-Nicolson scheme in the phase plane.

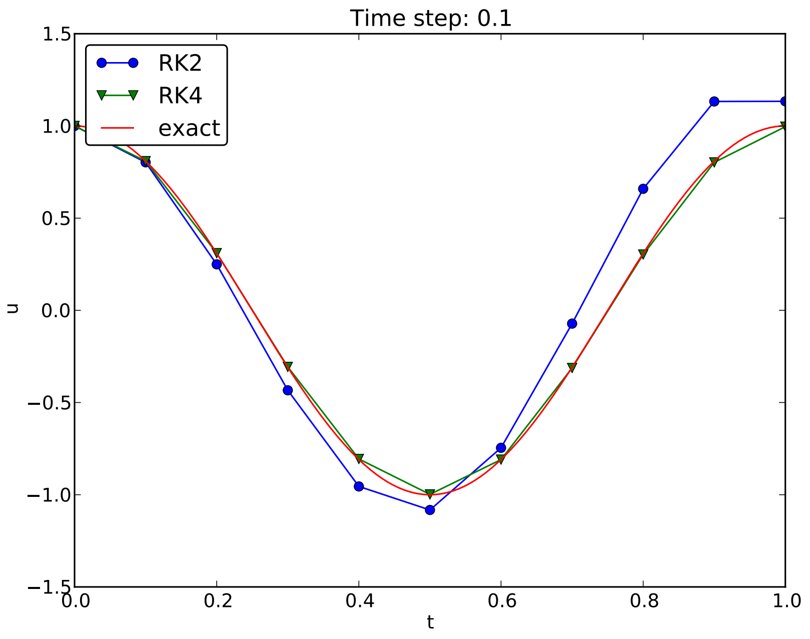

Figure 11: Long-time behavior of the Crank-Nicolson scheme.

3 Exercises

Exercise 1: Investigate phase errors

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

Exercise 2: Improve the accuracy by adjusting the frequency

According to 16, 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 `vb_undamped.py`) by `w = w*(1 - (1./24)*w**2*dt**2` and test how this adjustment in the numerical algorithm improves the accuracy. Filename: `vb_adjust_w.py`.

Exercise 3: Use a Taylor polynomial to compute u^1

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

Index