Study Guide: Vibration ODEs

Hans Petter Langtangen[1,2]

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

Sep 3, 2013

Contents

1   A simple vibration problem

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

Exact solution:

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

$u(t)$ oscillates with constant amplitude I and (angular) frequency ω. Period: $P = 2\pi/\omega$.

1.1   A centered finite difference scheme; step 1 and 2

- Strategy: follow the four steps[1] of the finite difference method.

- Step 1: Introduce a time mesh, here uniform on $[0, T]$: $t_n = n\Delta t$

---

[1] `http://tinyurl.com/k3sdbuv/pub/decay-sphinx/main_decay.html#the-forward-euler-scheme`

- Step 2: Let the ODE be satisfied at each mesh point:

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

1.2 A centered finite difference scheme; step 3

Step 3: Approximate derivative(s) by finite difference approximation(s). Very common (standard!) formula for u'':

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

Use this discrete initial condition together with the ODE at $t = 0$ to eliminate u^{-1} (insert (4) in (3)):

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

1.3 A centered finite difference scheme; step 4

Step 4: Formulate the computational algorithm. Assume u^{n-1} and u^n are known, solve for unknown u^{n+1}:

$$u^{n+1} = 2u^n - u^{n-1} - \omega^2 u^n.$$ (6)

Nick names for this scheme: Stormer's method or Verlet integration[2].

1.4 Computing the first step

- The formula breaks down for u^1 because u^{-1} is unknown and outside the mesh!

- And: we have not used the initial condition $u'(0) = 0$.

Discretize $u'(0)$ by a centered difference

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

Inserted in (6) for $n = 0$ gives

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

1.5 The computational algorithm

1. $u^0 = I$

2. compute u^1 from (8)

3. for $n = 1, 2, \ldots, N_t - 1$:

    (a) compute u^{n+1} from (6)

---

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

More precisly expressed in Python:

```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]
```

Note: `w` is consistently used for ω in code.

1.6   Operator notation; ODE

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

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

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

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

which is written out as

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

1.7   Operator notation; initial condition

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

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

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

1.8   Computing u'

u is often displacement/position, u' is velocity and can be computed by

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

2   Implementation

2.1   Core algorithm

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

def solver(I, w, dt, T):
    """
```

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

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

2.2   Plotting

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

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

2.3   Main program

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

2.4   User interface: 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
```

3 Verification

3.1 First steps for testing and debugging

- **Testing very simple solutions**: $u = \text{const}$ or $u = ct + d$ do not apply here (without a force term in the equation: $u'' + \omega^2 u = f$).

- **Hand calculations**: calculate u^1 and u^2 and compare with program.

3.2 Checking convergence rates

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, $E = \sqrt{\Delta t_i \sum_{n=0}^{N_t-1}(u^n - u_e(t_n))^2}$ in each case,

- estimates the rates r_i from two consecutive experiments $(\Delta t_{i-1}, E_{i-1})$ and $(\Delta t_i, E_i)$, assuming $E_i = C\Delta t_i^{r_i}$ and $E_{i-1} = C\Delta t_{i-1}^{r_i}$:

3.3 Implementational details

```
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
```

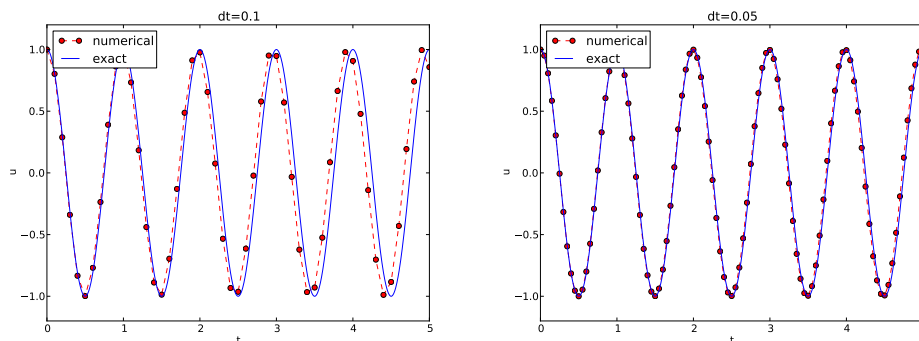Result: `r` contains values equal to 2.00 - as expected!

3.4 Nose test

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

```python
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)
```

Complete code in `vib_undamped.py`[3].

4 Long time simulations

4.1 Effect of the time step on long simulations

- 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 seems to grow with time.

4.2 Using a moving plot window

- In long time simulations we need a plot window that follows the solution.

- Method 1: `scitools.MovingPlotWindow`.

- Method 2: `scitools.avplotter` (ASCII vertical plotter).

Example:

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

Movie of the moving plot window[4].

---

[3]`http://tinyurl.com/jvzzcfn/vib/vib_undamped.py`
[4]`mov-vib/vib_undamped_dt0.05/index.html`

5 Analysis of the numerical scheme

5.1 Deriving an exact numerical solution; ideas

- Linear, homogeneous, difference equation for u^n.

- Has solutions $u^n \sim A^n$, where A is unknown (number).

- Here: $u_e(t) = I\cos(\omega t) \sim I\exp(i\omega t) = I(e^{i\omega\Delta t})^n$

- Trick for simplifying the algebra: $A = \exp(i\tilde{\omega}\Delta t)$ (ansatz)

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

- $\omega - \tilde{\omega}$ is the *phase error*

- Use the real part as the physical relevant part of a complex expression

5.2 Deriving an exact numerical solution; calculations (1)

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

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

5.3 Deriving an exact numerical; calculations (2)

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

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

Solve for $\tilde{\omega}$:

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

- Phase error because $\tilde{\omega} \neq \omega$.

- But how good is the approximation $\tilde{\omega}$ to ω?
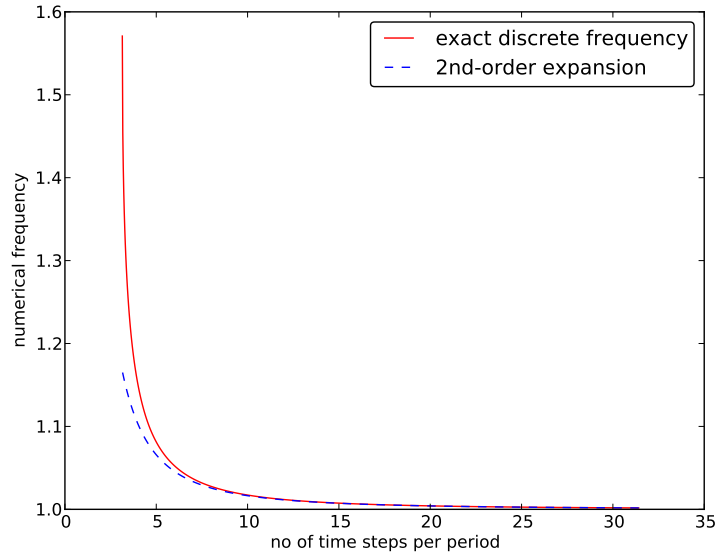
5.4 Polynomial approximation of the phase error

Taylor series expansion for small Δt gives a formula 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   # observe final /dt
```

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

The numerical frequency is too large (to fast oscillations).

5.5 Plot of the phase error

5.6 Exact discrete solution

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

Ideal for verification and analysis!

5.7 Stability

- Numerical solution has constant amplitude (desired!), but phase error.

- Constant amplitude requires $\sin^{-1}(\omega\Delta t/2)$ to be real-valued $\Rightarrow |\omega\Delta t/2| \leq 1$.

- $\sin^{-1}(x)$ is complex if $|x| > 1$, and then $\tilde{\omega}$ becomes complex.
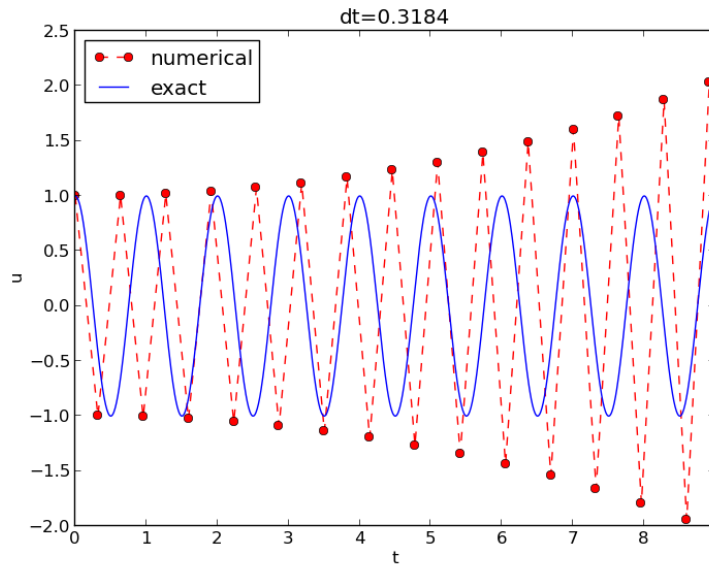
What is the consequence of complex $\tilde{\omega}$?

- Set $\tilde{\omega} = \tilde{\omega}_r + i\tilde{\omega}_i$.

- Since $\sin^{-1}(x)$ has a negative* imaginary part[5] for $x > 1$, $\exp(i\omega\tilde{t}) = \exp(-\tilde{\omega}_i t)\exp(i\tilde{\omega}_r t)$ leads to exponential growth $e^{-\tilde{\omega}_i t}$ when $-\tilde{\omega}_i t > 0$.

- This instability because the qualitative behavior is wrong.

5.8 The stability criterion

Cannot tolerate growth and must therefore demand a *stability criterion*

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

Try $\Delta t = \frac{2}{\omega} + 9.01 \cdot 10^{-5} \ (\pi^{-1}$:

5.9 Summary of the analysis

We can draw three important conclusions:

1. The key parameter in the formulas is $p = \omega\Delta t$.

---

[5]http://www.wolframalpha.com/input/?i=arcsin%28x%29%2C+x+in+%280%2C3%29

(a) Period of oscillations: $P = 2\pi/\omega$

(b) Number of time steps per period: $N_P = P/\Delta t$

(c) $\Rightarrow$ $p = \omega\Delta t = 2\pi N_P \sim N_P$

(d) The smallest possible N_P is $2 \Rightarrow p \in (0, \pi]$

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

3. u^n has a relative phase error $\tilde{\omega}/\omega \approx 1 + \frac{1}{24}p^2$, making numerical peaks occur too early

6 Alternative schemes based on 1st-order equations

6.1 Rewriting 2nd-order ODE as system of two 1st-order ODEs

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

Introduce auxiliary variable $v = u'$:

$$u' = v, \tag{19}$$
$$v' = -\omega^2 u. \tag{20}$$

Initial conditions: $u(0) = I$ and $v(0) = 0$.

6.2 The Forward Euler scheme

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

or written out,

$$u^{n+1} = u^n + \Delta t v^n, \tag{21}$$
$$v^{n+1} = v^n - \Delta t \omega^2 u^n. \tag{22}$$

6.3 The Backward Euler scheme

$$[D_t^- u = v]^{n+1}, \tag{23}$$
$$[D_t^- v = -\omega u]^{n+1}. \tag{24}$$

Written out:

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

This is a *coupled* 2×2 system for the new values at $t = t_{n+1}$!

11

6.4 The Crank-Nicolson scheme

$$[D_t u = \overline{v}^t]^{n+\frac{1}{2}}, \tag{27}$$

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

Also a coupled system:

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

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

6.5 Comparison of schemes via Odespy

Can use Odespy[6] to compare many methods for first-order schemes:

```
import odespy
import numpy as np

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

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

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

6.6 Forward and Backward Euler and Crank-Nicolson

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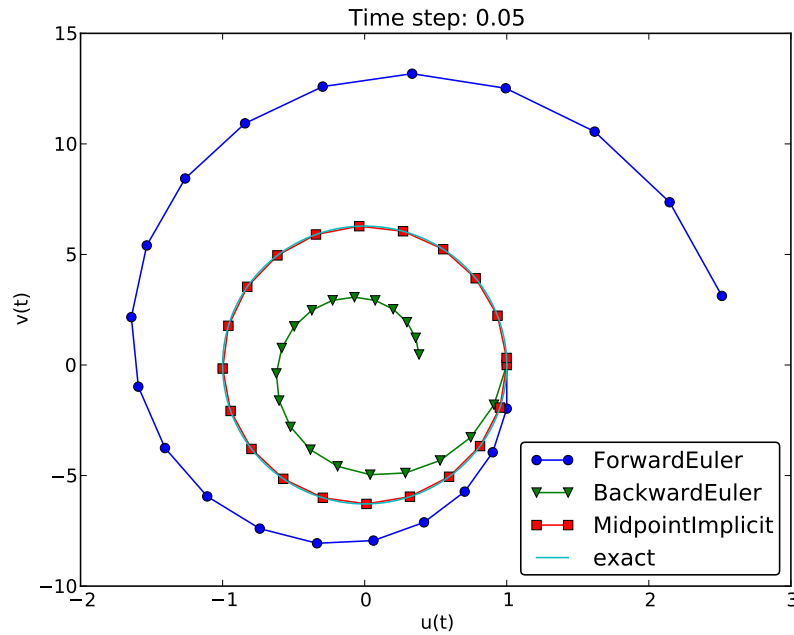
Two plot types:

- $u(t)$ vs t

---

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

- Parameterized curve $(u(t), v(t))$ in *phase space* (exact: $(I\cos\omega t, -\omega I\sin\omega t)$ = ellipse)

- Notice: The curve $(u(t), v(t))$ is closed and periodic $(P = 2\pi/\omega)$

6.7 Phase plane plot of the numerical solutions

6.8 Plain solution curves

6.9 Observations from the figures

- Forward Euler has growing amplitude and outward (u, v) spiral - pumps energy into the system.

- Backward Euler is opposite: decreasing amplitude, inward sprial, extracts energy.

- **Forward and Backward Euler are useless for vibrations.**
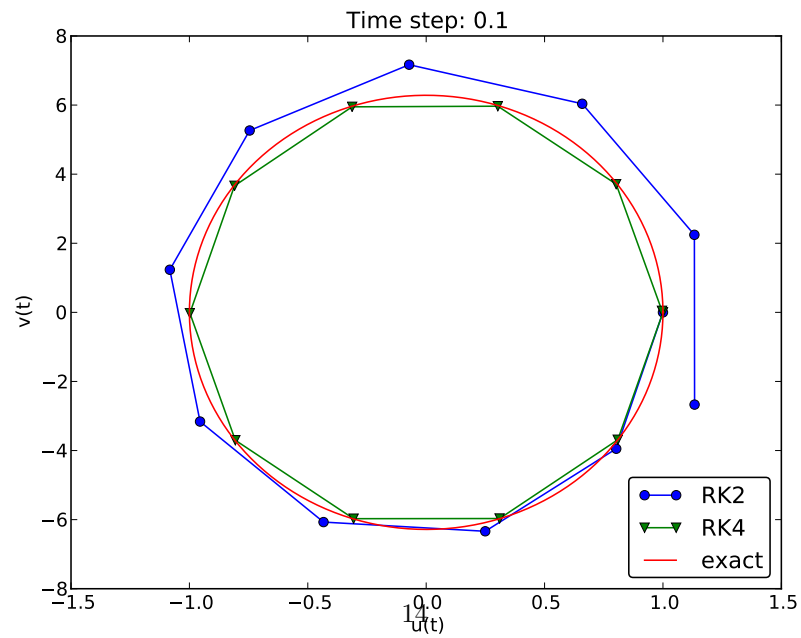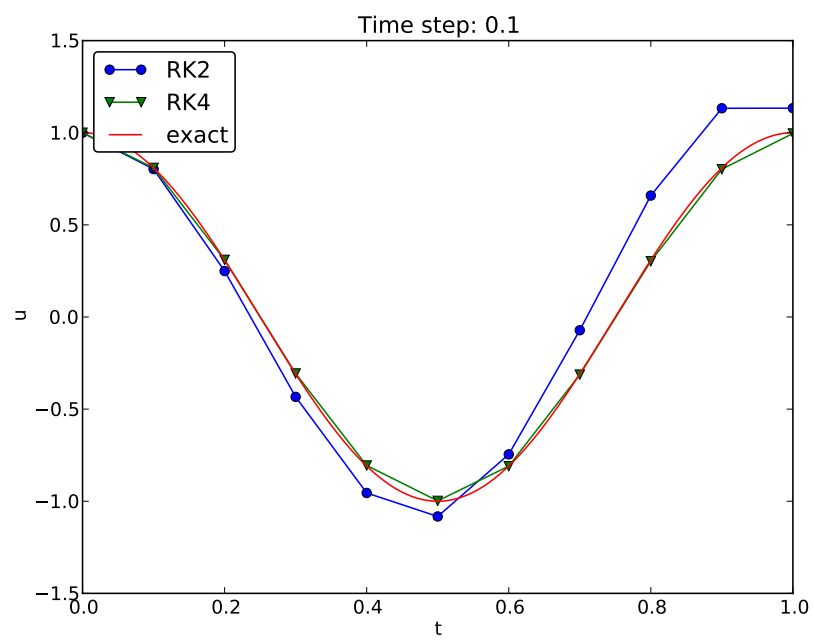
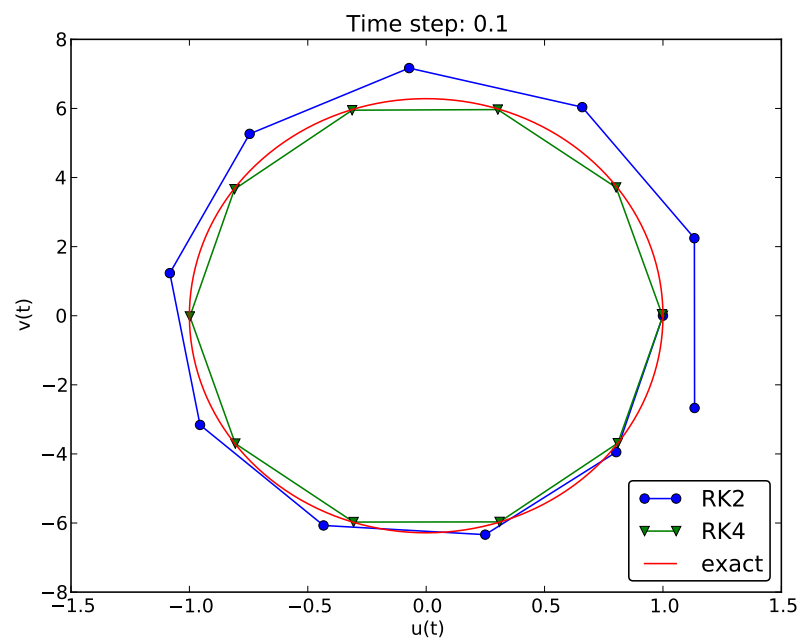- Crank-Nicolson (MidpointImplicit) looks much better.

13
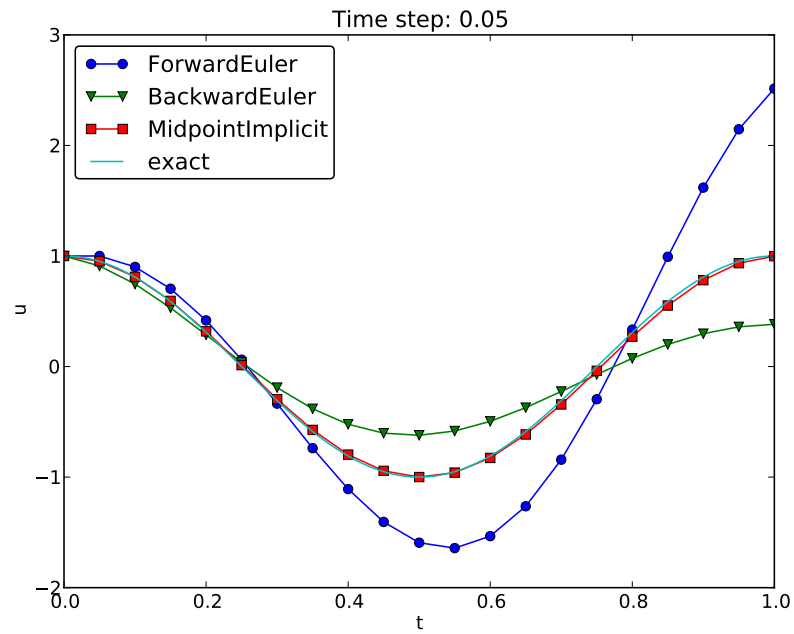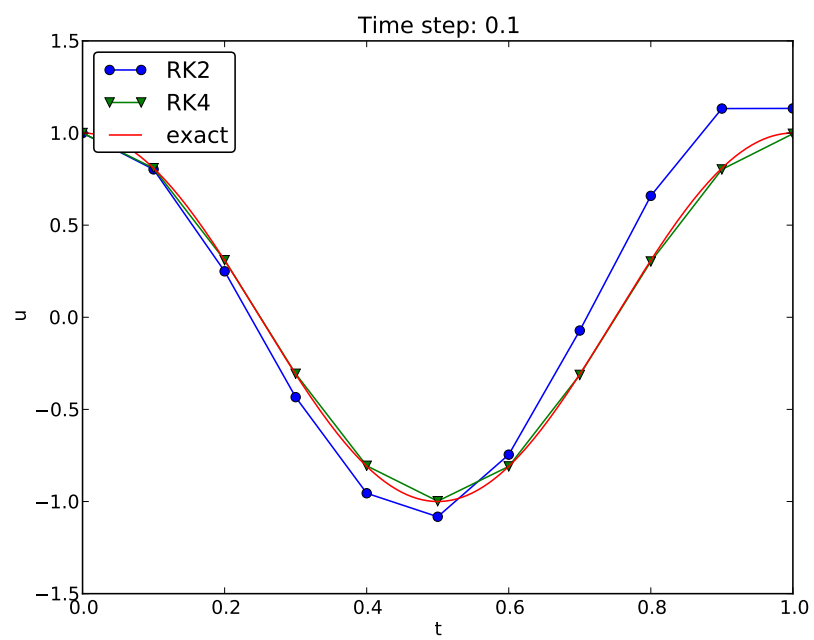
Figure 1: Comparison of classical schemes.

6.10 Runge-Kutta methods of order 2 and 4; short time series
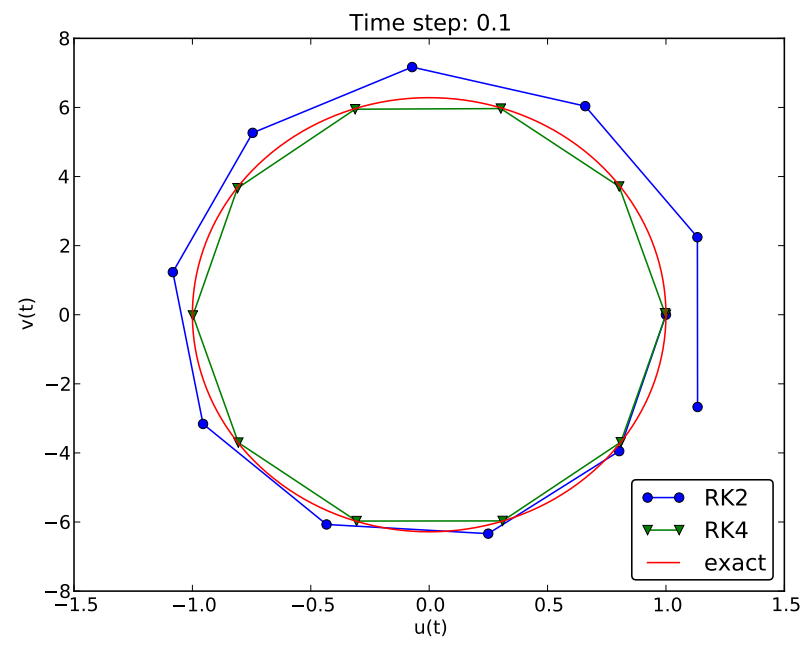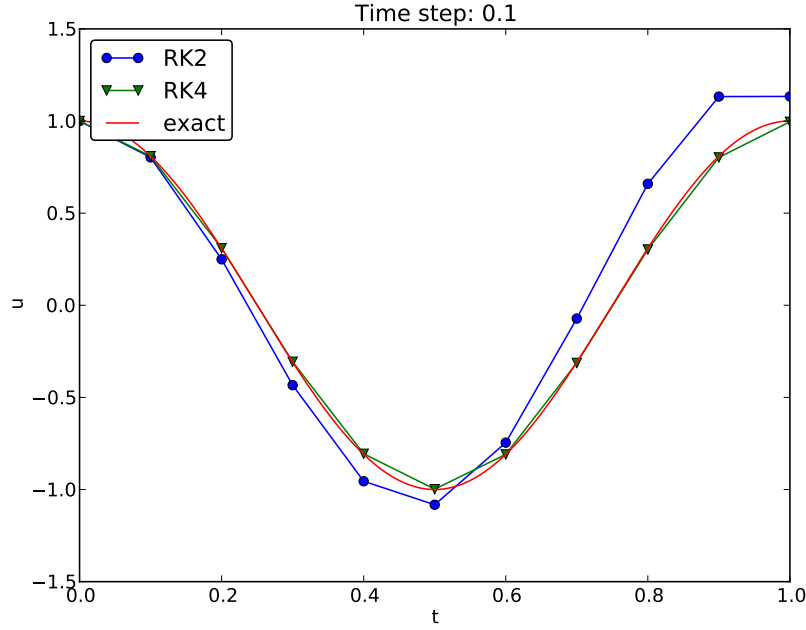
6.11   Runge-Kutta methods of order 2 and 4; longer time series

6.12 Crank-Nicolson; longer time series

(MidpointImplicit means CrankNicolson in Odespy)

6.13 Observations of RK and CN methods

- 4th-order Runge-Kutta is very accurate, also for large Δt.

- 2th-order Runge-Kutta is almost as bad as Forward and Backward Euler.

- Crank-Nicolson is accurate, but the amplitude is not as accurate as the difference scheme for $u'' + \omega^2 u = 0$.

6.14 The Euler-Cromer method

Forward-backward discretization:

- Update u with Forward Euler

- Update v with Backward Euler, using latest u

$$[D_t^+ u = v]^n, \tag{31}$$
$$[D_t^- v = -\omega u]^{n+1}. \tag{32}$$

Written out:

$$u^{n+1} = u^n + \Delta t v^n, \tag{33}$$
$$v^{n+1} = v^n - \Delta t \omega^2 u^{n+1}. \tag{34}$$

19

Names: Forward-backward scheme, Semi-implicit Euler method[7], symplectic Euler, semi-explicit Euler, Newton-Stormer-Verlet, and Euler-Cromer.

Forward Euler and Backward Euler are both $\mathcal{O}(\Delta t))$ approximations. What about the overall scheme? Expect $\mathcal{O}(\Delta t)$...

6.15 Equivalence with the scheme for the second-order ODE

Eliminate v^n:

From (34):

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

which can be inserted in (33) to yield

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

Using (33),

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

and when this is inserted in (35) we get

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

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 $u' = 0$:

$$u' = v = 0 \quad \Rightarrow \quad v^0 = 0,$$

and (33) implies $u^1 = u^0$, while (34) says $v^1 = -\omega^2 u^0$.

This $u^1 = u^0$ approximation corresponds to a first-order Forward Euler discretization of $u'(0) = 0$: $[D_t^+ u = 0]^0$.

---

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

Index