Verlet/leapfrog methods for damped systems

We assumed velocity-independent forces in leapfrog method;

\[ v_{n+1/2} = v_{n-1/2} + \Delta t a_n \]
\[ x_{n+1} = x_n + \Delta t v_{n+1/2} \]

With velocity dependent \( a(x, v, t) = F(x, v, x)/m \) we need \( v_n \) but have only \( v_{n+1/2} \)

To study this problem, separate damping from rest of force

\[ a(x, v, t) = \frac{1}{m} [F(x, t) - G(v)] \]

Consider approximation: \( a(x_n, v_n, t_n) \approx [F(x_n, t_n) - G(v_{n-1/2})]/m \)

\[ \hat{v}_{n+1/2} = v_{n-1/2} + \Delta_t [F(x_n, t_n) - G(v_{n-1/2})]/m \]
\[ \hat{x}_{n+1} = x_n + \Delta t \hat{v}_{n+1/2} \]

The error made in \( a \) is \( \sim \Delta t \) which gives x-error \( \sim \Delta_t^3 \)

We can do a second step using \( v_n = (\hat{x}_{n+1} - x_{n-1})/(2\Delta_t) \)

This renders the error in \( x \) \( \sim O(\Delta_t^4) \)
Summary; leapfrog algorithm with damping:

\[
\hat{v}_{n+1/2} = v_{n-1/2} + \Delta t \left[ F(x_n, t_n) - G(v_{n-1/2}) \right] / m \\
\hat{x}_{n+1} = x_n + \Delta t \hat{v}_{n+1/2} \\
v_n = (\hat{x}_{n+1} - x_{n-1}) / (2\Delta t) \\
v_{n+1/2} = v_{n-1/2} + \Delta t a_n \quad \text{v}_n \text{ used here in } a_n \\
x_{n+1} = x_n + \Delta t v_{n+1/2}
\]

Requires more work than standard leapfrog:

\[
v_{n+1/2} = v_{n-1/2} + \Delta t a_n \\
x_{n+1} = x_n + \Delta t v_{n+1/2}
\]
**Runge-Kutta method**

Classic high-order scheme; error $O(\Delta_t^5)$ (4th order)

Consider first single first-order equation: $\dot{x}(t) = f[x(t), t]$

**Warm-up: 2nd order Runge-Kutta**

Use mid-point rule:

$$\int_{t_n}^{t_{n+1}} f[x(t), t] dt = \Delta_t f[x(t_{n+1/2}), t_{n+1/2}] + O(\Delta_t^3)$$

But we don’t know $x(t_{n+1/2}) = x_{n+1/2}$

Approximate it using Euler formula;

$$\hat{x}_{n+1/2} = x_n + \frac{\Delta_t}{2} f(x_n, t_n) + O(\Delta_t^2)$$

Sufficient accuracy for $O(\Delta_t^3)$ formula:

$$x_{n+1} = x_n + \Delta_t f(\hat{x}_{n+1/2}, t_{n+1/2}) + O(\Delta_t^3)$$
4th-order Runge-Kutta (the real thing)

Uses Simpson’s formula: \( x_{n+1} = x_n + \frac{\Delta t}{6}(f_n + 4f_{n+1/2} + f_{n+1}) \)

Need to find \( O(\Delta_t^4) \) approximations for \( f_{n+1/2}, f_{n+1} \)

Somewhat obscure scheme accomplishes this (can be proven correct using Taylor expansion)

\[
\begin{align*}
\hat{x}_{n+1/2} &= x_n + \Delta t f(x_n, t_n)/2 \\
\hat{x}'_{n+1/2} &= x_n + \Delta t f(\hat{x}_{n+1/2}, t_{n+1/2})/2
\end{align*}
\]

\[
\begin{align*}
k_1 &= \Delta t f(x_n, t_n) \\
k_2 &= \Delta t f(x_n + k_1/2, t_{n+1/2}) \\
k_3 &= \Delta t f(x_n + k_2/2, t_{n+1/2}) \\
k_4 &= \Delta t f(x_n + k_3, t_{n+1})
\end{align*}
\]

\[x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\]
Runge-Kutta for two coupled equations

\[ \dot{x}(t) = f(x, y, t) \quad \dot{y}(t) = g(x, y, t) \]

\begin{align*}
k_1 &= \Delta_t f(x_n, y_n, t_n), \\
l_1 &= \Delta_t g(x_n, y_n, t_n), \\
k_2 &= \Delta_t f(x_n + k_1/2, y_n + l_1/2, t_{n+1/2}), \\
j_2 &= \Delta_t g(x_n + k_1/2, y_n + l_1/2, t_{n+1/2}), \\
k_3 &= \Delta_t f(x_n + k_2/2, y_n + l_2/2, t_{n+1/2}), \\
l_3 &= \Delta_t g(x_n + k_2/2, y_n + l_2/2, t_{n+1/2}), \\
k_4 &= \Delta_t f(x_n + k_3, y_n + l_3, t_{n+1}), \\
l_4 &= \Delta_t g(x_n + k_3, y_n + l_3, t_{n+1}), \\
x_{n+1} &= x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \\
y_{n+1} &= y_n + \frac{1}{6}(l_1 + 2l_2 + 2l_3 + l_4),
\end{align*}
Equations of motion, Runge-Kutta algorithm

\[ k_1 = \Delta t a(x_n, v_n, t_n), \]
\[ l_1 = \Delta t v_n, \]
\[ k_2 = \Delta t a(x_n + l_1/2, v_n + k_1/2, t_{n+1/2}), \]
\[ l_2 = \Delta t (v_n + k_1/2), \]
\[ k_3 = \Delta t a(x_n + l_2/2, v_n + k_2/2, t_{n+1/2}), \]
\[ l_3 = \Delta t (v_n + k_2/2), \]
\[ k_4 = \Delta t a(x_n + l_3, v_n + k_3, t_{n+1}), \]
\[ l_4 = \Delta t (v_n + k_3), \]
\[ v_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \]
\[ x_{n+1} = x_n + \frac{1}{6}(l_1 + 2l_2 + 2l_3 + l_4). \]

Including damping is no problem here
The RK method does not have time-reversal symmetry

- Errors not bounded for periodic motion
- Time-reversibility important in some applications

Advantages of RK relative to leapfrog:

- Variable time-step can be used (uses only n-data for n+1)
- Better error scaling (but more computations for each step)
What algorithm to use?

**Recommendation**

In the case of energy-conserving systems (no damping or external driving forces)
- **Use the Verlet/leapfrog algorithm**
  - good energy-conserving property (no divergence)

In the case of non-energy-conserving systems (including damping and/or external driving forces)
- Energy is not conserved, so no advantage for Verlet
- **Use the Runge-Kutta algorithm**
  - smaller discretization error for given integration time $T$
Motion in more than one dimension

Vector equations of motion
\[ \ddot{x}(t) = \ddot{v}(t) \]
\[ \ddot{v}(t) = \frac{1}{m} \vec{F}[\vec{x}(t), \vec{v}(t), t] \]

Different components (dimensions) coupled through F

Example: Planetary motion (in a 2D plane)

Gravitational force:
\[ \vec{F}(r) = -\frac{GMm}{r^3} \cdot \vec{r} \]

Two-body problem; can be reduced to one-body problem for effective mass:
\[ \mu = \frac{Mm}{(m + M)} \]

Consider \( M >> m \), assume \( M \) stationary
Equations of motion for the x and y coordinates

\[ \begin{align*}
\dot{x} &= v_x \\
\dot{v}_x &= -\frac{GMx}{r^3} \\
\dot{v}_y &= -\frac{GMy}{r^3} \\
\dot{y} &= v_y
\end{align*} \quad r = \sqrt{x^2 + y^2} \]

The leapfrog algorithm is

\[ \begin{align*}
x(n+1) &= x(n) + \Delta t v_x(n + 1/2) \\
y(n+1) &= y(n) + \Delta t v_y(n + 1/2) \\
v_x(n + 1/2) &= v_x(n - 1/2) - \Delta t GMx(n)[x^2(n) + y^2(n)]^{-3/2} \\
v_y(n + 1/2) &= v_y(n - 1/2) - \Delta t GMy(n)[x^2(n) + y^2(n)]^{-3/2}
\end{align*} \]

Not much harder than 1D

Runge-Kutta also easily generalizes to D>1
Program example: de-orbiting a satellite

Program ‘crash.f90’ on course web site:

- Solves equations of motion for a satellite, including forces of
  - gravitation
  - atmospheric drag
  - thrust of rocket motor
  for de-orbiting

- We know gravitational force
- Rocket motor causes a constant deceleration for limited (given) time
- We will create a model for air drag

\[ F = F_{\text{gravity}}(r)\vec{e}_r + F_{\text{rocket}}(t)\vec{e}_v + F_{\text{drag}}(r, v)\vec{e}_v \]
Gravitation

\[
\frac{\vec{F}_{\text{gravity}}}{m} = \frac{GM}{r^2} \vec{e}_r
\]

Braking using rocket motor during given time, starting at \( t=0 \)

\[
\frac{\vec{F}_{\text{rocket}}}{m} = \Theta(T_{\text{brake}} - t) B \vec{e}_v
\]

Assuming constant deceleration \( B \), e.g., \( B=5 \text{ m/s}^2 \)

Atmospheric drag; depends on density of air

\[
\frac{\vec{F}_{\text{drag}}}{m} = \frac{C_d}{m} \rho(h) v^2 \vec{e}_v
\]
\[\rho(0) \approx 1.2\text{ kg/m}^3\]

Adjusting drag-coefficient to give reasonable terminal velocity

\[
\frac{F_{\text{drag}}}{m} = g \rightarrow \frac{C_d}{m} = \frac{g}{\rho(h) v_t^2}
\]
(at \( h=0 \))

\[
\frac{C_d}{m} = 8 \cdot 10^{-4} \frac{m^2}{\text{kg}}
\]
gives \( v_t \approx 100\text{ m/s} \)
Model for the atmospheric density

This form turns out to give good agreement with data:

$$\rho(h) = 1.225 \cdot \exp \left[ - \left( \frac{h}{k_1} + \left( \frac{h}{k_{3/2}} \right)^{3/2} \right) \right] \text{ kg/m}^3$$

$$k_1 = 1.2 \cdot 10^4 \text{ m} \text{ and } k_2 = 2.2 \cdot 10^4 \text{ m}$$

Difficult to model atmosphere > 40 km

Let’s see what the model gives for the stability of low orbits
Some elements of the program crash.f90

Parameters and some variables in module systemparam

module systemparam

    real(8), parameter :: pi=3.141592653589793d0
    real(8), parameter :: gm=3.987d14  ! G times M of earth
    real(8), parameter :: arocket=5.d0 ! Deceleration due to engine
    real(8), parameter :: dragc=8.d-4  ! Air drag coefficient / m
    real(8), parameter :: re=6.378d6   ! Earth's radius

    real(8) :: dt,dt2  ! time step, half of the time step
    real(8) :: tbrake  ! run-time of rocket engine

end module systemparam

All program units including the statement
    use systemparam
can access these constants and variables
Main program

Reads input data from the user:

```fortran
print*,'Initial altitude of satellite (km)'; read*,r0
r0=r0*1.d3+re
print*,'Rocket run-time (seconds)'; read*,tbrake
print*,'Time step delta-t for RK integration (seconds)';read*,dt
dt2=dt/2.d0
print*,'Writing results every Nth step; give N';read*,wstp
print*,'Maximum integration time (hours)';read*,tmax
tmax=tmax*3600.d0
```

Sets initial conditions:

```fortran
x=r0
y=0.d0
vx=0.d0
vy=sqrt(gm/r0)
nstp=int(tmax/dt)
```

Velocity of object in a Kepler orbit of radius \( r \):

\[
v = \sqrt{\frac{GM_e}{r}}
\]

Opens a file to which data will be written

```fortran
open(1,file='sat.dat',status='replace')
```
Main loop for integrations steps:

```fortran
do i=0,nstp
    call polarposition(x,y,r,a)
    if (r > re) then
        t=dble(i)*dt
        if(mod(i,wstp)==0)write(1,1)t,a,(r-re)/1.d3,sqrt(vx**2+vy**2)
        1 format(f12.3,' ',f12.8,' ',f14.6,' ',f12.4)
        call rkstep(t,x,y,vx,vy)
    else
        print*,'The satellite has successfully crashed!'
        goto 2
    end if
end do
go to 2
print*,'The satellite did not crash within the specified time.'
close(1)
```

Polar coordinates from subroutine `polarposition(x,y,r,a)`

```fortran
r=sqrt(x**2+y**2)
if (y >= 0.d0) then
    a=acos(x/r)/(2.d0*pi)
else
    a=1.d0-acos(x/r)/(2.d0*pi)
end if
```
Runge-Kutta integration step by \texttt{rkstep(t0, x0, y0, vx0, vy0)}

\begin{verbatim}
t1=t0+dt; \ th=t0+dt2
call accel(x0,y0,vx0,vy0,t0,ax,ay)
kx1=dt2*ax
ky1=dt2*ay
lx1=dt2*vx0
ly1=dt2*vy0
call accel(x0+lx1,y0+ly1,vx0+kx1,vy0+ky1,th,ax,ay)
kx2=dt2*ax; ky2=dt2*ay
lx2=dt2*(vx0+kx1)
ly2=dt2*(vy0+ky1)
call accel(x0+lx2,y0+ly2,vx0+kx2,vy0+ky2,th,ax,ay)
kx3=dt*ax
ky3=dt*ay
lx3=dt*(vx0+kx2)
ly3=dt*(vy0+ky2)
call accel(x0+lx3,y0+ly3,vx0+kx3,vy0+ky3,t1,ax,ay)
kx4=dt2*ax
ky4=dt2*ay
lx4=dt2*(vx0+kx3)
ly4=dt2*(vy0+ky3)
x1=x0+(lx1+2.d0*lx2+lx3+lx4)/3.d0
y1=y0+(ly1+2.d0*ly2+ly3+ly4)/3.d0
vx1=vx0+(kx1+2.d0*kx2+kx3+kx4)/3.d0
vy1=vy0+(ky1+2.d0*ky2+ky3+ky4)/3.d0
x0=x1; y0=y1; vx0=vx1; vy0=vy1
\end{verbatim}
Accelarations calculated in \texttt{accel(x,y,vx,vy,t,ax,ay)}

\begin{align*}
r &= \sqrt{x^2 + y^2} \\
v^2 &= vx^2 + vy^2 \\
v^1 &= \sqrt{v^2} \\
r^3 &= 1.d0 / r^3 \\
ax &= -gm * x * r^3 \\
ay &= -gm * y * r^3 \\
end{align*}

\textbf{!!!} evaluates the acceleration due to gravitation

\begin{align*}
\text{if} \ (v^1 > 1.d-12) \ \text{then} \\
\text{ad} &= \text{dragc} * \text{airdens}(r) * v^2 \\
ax &= ax - ad * vx / v^1 \\
ay &= ay - ad * vy / v^1 \\
\text{endif}
\end{align*}

\textbf{!!!} evaluates the acceleration due to air drag

\begin{align*}
\text{if} \ (t < tbrake . \text{and.} \ v^1 > 1.d-12) \ \text{then} \\
ax &= ax - arocket * vx / v^1 \\
ay &= ay - arocket * vy / v^1 \\
\text{endif}
\end{align*}

\textbf{!!!} evaluates the acceleration due to rocket motor thrust
Let’s play with the program in various ways...

Start with \( v=0 \) (change in program); the satellite drops like a rock

What happens as it enters the atmosphere?

The atmosphere becomes important at 40-50 km altitude.
The actual final velocity is 102.6 m/s (exactly 100 m/s terminal velocity results when assuming constant sea-level air density).
Looking at low-altitude orbits without starting the motor
Atmospheric drag brings down the satellite for $h < 200$ km
Starting from 100 km, the satellite completes just 1 revolution
(relative air density is $1.5 \times 10^{-8}$ of sea-level density at 100 km)

Velocity little changed until 40 km altitude (direction changes)
Starting from 200 km, the satellite doesn’t come down
(model likely has too little atmosphere for $> 120$ km)
Starting from 120 km; crash in 88 hours

Path sampled every 30 s (integrated using $dt = 1$ s)
Finally, let’s turn on the rocket motor; start at 200 km

Running the motor for 5, 10, 20 seconds

Braking for 5 s at -5 m/s² is not enough to bring it down during first revolution; many almost elliptic orbits before crash