Ch. 07
Ordinary Differential Equations (ODE)
[Initial Value Problems]

Andrea Mignone
Physics Department, University of Torino
AA 2018-2019

Ordinary Differential Equations (ODE)

- Many physical problems are described by **O**rdinary **D**ifferential **E**quations (*ODE*) and therefore it is not surprising that this particular field plays a major role in computational physics

- A very common form is

$$\frac{d\vec{Y}}{dt} = \vec{R}(t, \vec{Y}) \qquad \text{with} \quad \vec{Y}(t=0) = \vec{Y}_0$$

where $\vec{Y} \in \mathbb{R}^n$ and t is the independent variable. In components:

$$\frac{dY_i}{dt} = R_i(t, \vec{Y}) \quad \text{for} \quad i = 1, .., n$$

- This defines an **I**nitial **V**alue **P**roblem (*IVP*): an ODE together with a specified value, called the initial condition, of the unknown function at a given point in the domain of the solution.

Ordinary Differential Equations

- Another example is the **B**oundary **V**alue **P**roblem (BVP): a differential equation together with a set of additional constraints, called the boundary conditions.

- The "standard" two point boundary value problem has the following form:

$$\frac{d\vec{Y}}{dt} = \vec{R}(t, \vec{Y}) \qquad \text{with} \qquad \begin{cases} Y_i(0) = \alpha_i & \text{for} \quad i = 1, ..., m \\ Y_j(t_e) = \beta_j & \text{for} \quad j = m+1, ...n \end{cases}$$

- → We want to solve a set of n coupled first-order ordinary differential equations, satisfying m boundary conditions at the starting point (t=0), and a remaining set of $n-m$ boundary conditions at the final point t_e.

Higher order ODE

- A differential equation of order higher than one can always be rewritten as a coupled set of first-order ODEs.

- As an example,

$$\frac{d^2z}{dt^2} = F(z,t) \quad \Longrightarrow \quad \begin{cases} \dfrac{dY_0}{dt} = Y_1 \\ \dfrac{dY_1}{dt} = F(z,t) \end{cases}$$

where $Y_0 = z$, $Y_1 = z'$.

- Occasionally, it is useful to incorporate into their definition some other factors in the equation, or some powers of the independent variable, for the purpose of mitigating singular behavior that could result in overflows or increased roundoff error.

Initial Value Problem

- In the following we discuss several methods for solving ODE with a particular emphasis on the *initial value problem*, consisting of the solution of the original ODE once the vector of unknowns Y is specified at a particular time:

$$\frac{d\vec{Y}}{dt} = \vec{R} \qquad \text{with} \quad \vec{Y}_0 = \vec{Y}(t = t_0)$$

  where Y_0 is the *initial condition*.
- This is a very common case in physics.

- We consider an interval `[0,te]` which represents our domain of integration.

- The general strategy is to divide the interval into steps of equal size $h = t_e/N$ and then develop a recursion formula relating $\mathbf{Y}_n$ with $\mathbf{Y}_{n-1}$, $\mathbf{Y}_{n-2}$,... and so forth.

- Here $\mathbf{Y}_n$ is the solution vector at $t_n = nh$.

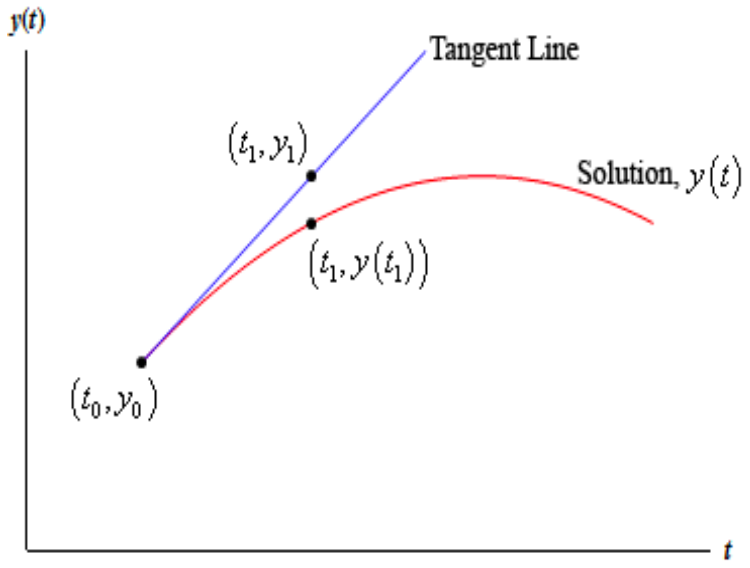
The Euler Method

- The simplest method is the explicit Euler method in which we replace the temporal derivative by finite differences:

$$\frac{dY}{dt} \approx \frac{Y_{n+1} - Y_n}{h} + O(h) = R(t_n, Y_n)$$

which gives

$$Y_{n+1} = Y_n + hR(t_n, Y_n) + O(h^2)$$



- This formula has a *local error* of h^2.

- However, the *global truncation* error after many steps is the cumulative effect of the local errors committed at each step:

$$NO(h^2) \approx O(h)$$

- In this sense, the Euler method is therefore a 1st order method.

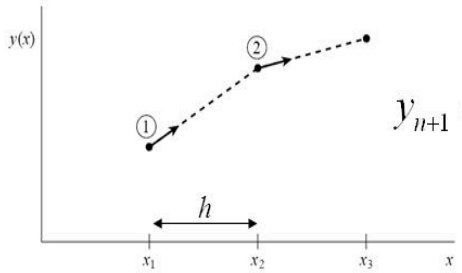
Order of Accuracy

- The method has order of accuracy p if the local truncation error satisfies

$$\epsilon = O(h^{p+1})$$

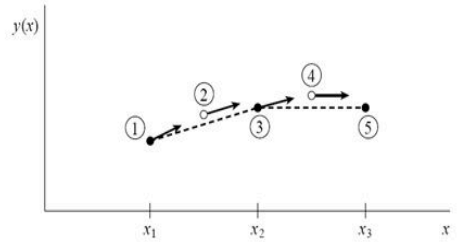
- The order of accuracy is therefore defined to be one less than the order of the leading term in the local truncation error.

Euler and Midpoint



$$y_{n+1} = y_n + h f(x_n, y_n) + O(h^2)$$

Figure 16.1.1. Euler's method. In this simplest (and least accurate) method for integrating an ODE, the derivative at the starting point of each interval is extrapolated to find the next function value. The method has first-order accuracy.



$$k_1 = h f(x_n, y_n)$$

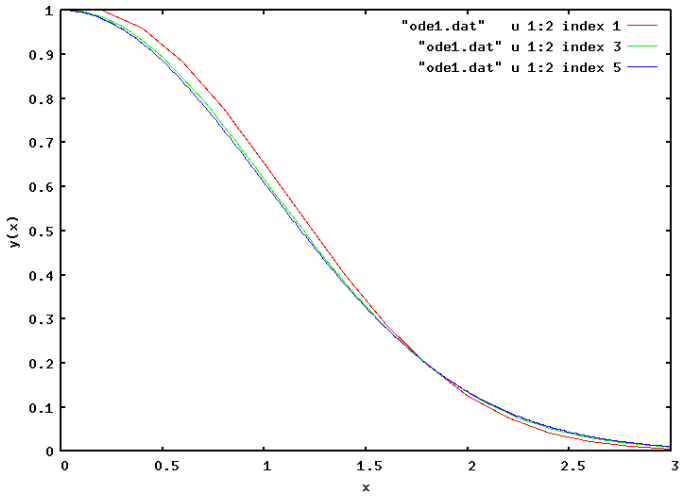
$$k_2 = h f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1)$$

$$y_{n+1} = y_n + k_2 + O(h^3)$$

Figure 16.1.2. Midpoint method. Second-order accuracy is obtained by using the initial derivative at each step to find a point halfway across the interval, then using the midpoint derivative across the full width of the interval. In the figure, filled dots represent final function values, while open dots represent function values that are discarded once their derivatives have been calculated and used.

Practice Session #1

- Consider the ODE $\frac{dy}{dx} = -xy\,; \quad y(0) = 1$



This has analytical solution $y = \exp(-x^2/2)$

→ Implement Euler's method and try integrating from x = 0 up to x = 3 using different step sizes, e.g., `h = 0.5, 0.2, 0.1, 0.05, ..., 0.`

The following table shows the numerical solution and errors for `h = 0.5`.

# x	y(x)	abs_err	rel_err
0.000000e+00	1.000000e+00	0.000000e+00	0.000000e+00
5.000000e-01	1.000000e+00	1.175031e-01	1.331485e-01
1.000000e+00	7.500000e-01	1.434693e-01	2.365410e-01
1.500000e+00	3.750000e-01	5.034753e-02	1.550813e-01
2.000000e+00	9.375000e-02	4.158528e-02	3.072760e-01
2.500000e+00	0.000000e+00	4.393693e-02	1.000000e+00
3.000000e+00	0.000000e+00	1.110900e-02	1.000000e+00

→ Write ASCII data file showing the absolute and relative errors for each step size as a function of position. Comment on this.

→ Choose `h = 1`. What happens ?

Code Structure

- An efficient structure can be achieved by separating the implementation of the actual integrator (which is *general*) from the right hand side function (which is *problem-dependent*):

```
void EulerStep (double t, double *Y, void (*dYdt)(double, double *, double *),
                double dt, int neq)
// Take one step dt using Euler method for the solution of dY/dt = rhs.
// Here neq is the number of ODE (the dimensionality of Y[]) and *dYdt() is a
// pointer to the function that calculates the right hand side of the system
// of equations.
{
  int n;
  double rhs[neq];

  dYdt (t, Y, rhs);
  for (n = 0; n < neq; n++){
    Y[n] += dt*rhs[n];
  }
}
```

```
void dYdt (double t, double *Y, double *R)
// Compute the right hand side of the ODE dy/dt = -t*y
{
  double y = Y[0];
  R[0] = -t*y;
}
```

Achieving Higher Accuracy

- Integrating an ODE is closely related to numerical quadrature. In fact, we can formally integrate the ODE and rewrite it as

$$Y_{n+1} = Y_n + \int_{t_n}^{t_{n+1}} R(t, Y)\, dt$$

- The problem, of course, is that we don't know the right hand side R over the interval of integration.

- Two main approaches can be used to achieve higher order:

  - *Multistep methods*: achieve higher order by considering solution values further in the past (Y_n, Y_{n-1}, Y_{n-2}, etc…)
  - *Single-Step methods*: propagate a solution over an interval by combining the information from several Euler-style steps (each involving one evaluation of Rs), and then using the information obtained to match a Taylor series expansion up to some higher order.

Linear Multistep Methods

- A possible solution is to use the values at previous point to provide linear extrapolation of R over the interval

$$R \approx \frac{t - t_{n-1}}{h} R_n - \frac{t - t_n}{h} R_{n-1} + O(h^2)$$

- Doing the integral results in the *Adams-Bashforth* two-step method:

$$Y_{n+1} = Y_n + h\left(\frac{3}{2} R_n - \frac{1}{2} R_{n-1}\right) + O(h^2)$$

- A related method is that of *Adam-Moulton* that employs a trapezoidal rule:

$$Y_{n+1} = Y_n + h\left(\frac{R_n + R_{n+1}}{2}\right) + O(h^2)$$

This method, however, is implicit since the unknown appears also in the r.h.s through R_{n+1}. In the following we will focus mainly in explicit methods

Linear Multistep Methods

- Extrapolating using higher-order polynomial allows us to achieve higher accuracy.

- The fourth-order *Adams-Bashforth* method can be derived using a cubic interpolant:

$$Y_{n+1} = Y_n + \frac{h}{24}\left(55R_n - 59R_{n-1} + 37R_{n-2} - 9R_{n-3}\right) + O(h^5)$$

- Since the recursion involve several previous steps, the value of Y_0 alone is not sufficient to start and some other method is required to start the integration (e.g. RK or Taylor-series methods).

Runge-Kutta Methods

- Runge-Kutta methods are single-step methods as they require information only in the current interval and not from previous ones.

- They are based on Taylor expansion and yield better algorithms in terms of accuracy and stability.

- The basic philosophy is that it provides intermediate step in the computation of Y^{n+1}. As an example we consider the 2nd order RK method where we approximate the integral using midpoint rule:

$$Y_{n+1} = Y_n + hR(t_{n+1/2}, y_{n+1/2}) + O(h^3)$$

- To evaluate the function at the midpoint on the r.h.s. we can use a lower-order approximation:

$$\begin{cases} Y_* &= Y_n + \frac{h}{2}R(t_n, Y_n) & \leftarrow \text{predictor} \\ Y_{n+1} &= Y_n + hR(t_{n+1/2}, Y_*) & \leftarrow \text{corrector} \end{cases}$$

Runge-Kutta Methods

- Similarly one could use a trapezoidal (instead of the midpoint) rule:

$$Y_{n+1} = Y_n + \frac{h}{2}\Big[R(t_n, Y_n) + R(t_{n+1}, Y_{n+1})\Big] + O(h^3)$$

- However, to make the algorithm explicit, we replace the second term in the r.h.s. with a lower-order approximation using Euler's method:

$$\begin{cases} Y_* &= Y_n + hR(t_n, Y_n) + O(h^2) & \leftarrow \text{predictor} \\ Y_{n+1} &= Y_n + \frac{h}{2}\Big[R(t_n, Y_n) + R(t_{n+1}, Y_*)\Big] + O(h^3) & \leftarrow \text{corrector} \end{cases}$$

- This method is second-order accurate in time and it is also known as Heun's method or *modified Euler's method*.

RK2 Methods

- Runge-Kutta methods are traditionally written in terms of increments, so that the midpoint and the modified Euler's method can be written as:

$$\begin{cases} k_1 &= R(t_n, Y_n) \\ k_2 &= R\left(t_n + \frac{h}{2}, Y_n + \frac{h}{2}k_1\right) \\ Y_{n+1} &= Y_n + hk_2 + O(h^3) \end{cases} \qquad \text{[RK midpoint]}$$

$$\begin{cases} k_1 &= R(t_n, Y_n) \\ k_2 &= R\left(t_n + h, Y_n + hk_1\right) \\ Y_{n+1} &= Y_n + h\frac{k_1 + k_2}{2} + O(h^3) \end{cases} \qquad \text{[Modified Euler]}$$

Practical Implementation of RK Methods

- From a practical view, here's a pseudo-code showing how the RK2 (midpoint) can be implemented for a system of `Neq` equations:

$$k_1 = R(t_n, Y_n)$$

$$k_2 = R\left(t_n + \frac{h}{2}, Y_n + \frac{h}{2}k_1\right)$$

$$Y_{n+1} = Y_n + hk_2 + O(h^3)$$

```
RK2Step(t, Y, h, Neq, (*RHS_Func))
{
  double Y1[Neq], k1[Neq], k2[Neq]

  RHS_Func(t,Y,k1)
  for (i=1..Neq) Y1[i] = Y[i]+0.5*h*k1[i]

  RHS_Func(t+0.5*h,Y1,k2)

  for (i=1..Neq) Y[i] += h*k2[i]
}
```

- Here `RHS_Func(double, double *, double *)` is a *user-supplied* function that *depends on the particular problem* and whose purpose is that of *computing the array of right hand sides* of the first-order ODEs.

- `RHS_Func()` takes two input values (`t`, `Y`) and one output value (k_n).

The fourth-order Runge-Kutta: RK4

- If the Improved Euler method for differential equations corresponds to the Trapezoid Rule for numerical integration, we might look for an even better method corresponding to Simpson's Rule. This is called the Fourth-Order Runge-Kutta Method.

- The classical and very popular fourth-order algorithm requires 4 evaluation of function per step, has local accuracy of $O(h^5)$ and has found to give the best balance between accuracy and computational effort:

$$
\begin{aligned}
k_1 &= R(t_n, Y_n) \\
k_2 &= R(t_n + \tfrac{h}{2}, Y_n + \tfrac{h}{2}k_1) \\
k_3 &= R(t_n + \tfrac{h}{2}, Y_n + \tfrac{h}{2}k_2) \\
k_4 &= R(t_n + h, Y_n + hk_3) \\
Y_{n+1} &= Y_n + \tfrac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)
\end{aligned}
$$

Practice Session #2

- `ode2.cpp`: using Euler, RK2 and RK4 method, solve the system of ODEs

$$\begin{cases} \dot{x} = y \\ \dot{y} = -x \end{cases}$$

with initial condition `x(0) = 1, y(0) = 0.`

→ What is the analytical solution ?

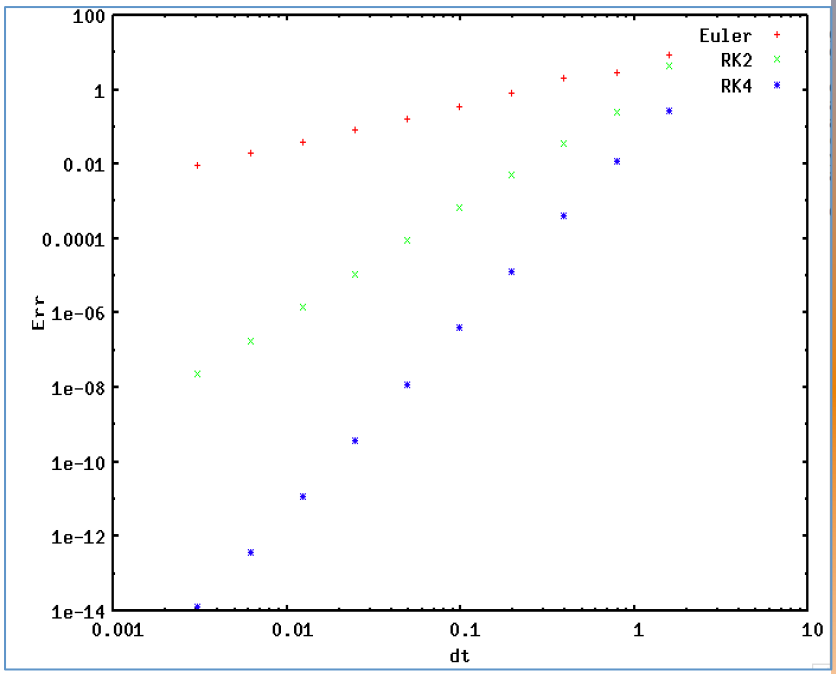
→ Try to integrate between 0 ≤ t ≤ 20π using 200 points and compare the solutions. Which method is the most accurate ?

→ How can you choose the step size ?

→ Is there any quantity that you expect to be conserved ? Are they ?

Practice Session #2: Convergence study

- We now wish to measure the convergence study of the three different algorithms (Euler, RK2, RK4)



- Integrate now the previous system of ODE between $0 \leq t \leq 2\pi$ using N = 4, 8, 16, … 2048 intervals.

- At the end of integration compute the error as the difference between numerical and analytical solutions:

```
err = |x(N-1) – cos(t[n-1])|;
```

- Plot the error as a function of N. Do you recover the expected convergence rate ?

Example We study how Euler's method behaves for the stable model problem above, i.e., in the case $\lambda \leq 0$. Since $f(t, y) = \lambda y(t)$ Euler's method states that

$$\begin{aligned} y_{n+1} &= y_n + hf(t_n, y_n) \\ &= y_n + h\lambda y_n \\ &= (1 + \lambda h) y_n. \end{aligned}$$

Therefore, by induction,

$$y_n = (1 + \lambda h)^n y_0.$$

Since the exact problem has an exponentially decaying solution for $\lambda < 0$, a stable numerical method should exhibit the same behavior. Therefore, in order to ensure stability of Euler's method we need that the so-called *growth factor* $|1 + \lambda h| < 1$. For real $\lambda < 0$ this is equivalent to

$$-2 < h\lambda < 0 \quad \Longleftrightarrow \quad h < \frac{-2}{\lambda}.$$

Thus, Euler's method is only *conditionally stable*, i.e., the step size has to be chosen sufficiently small to ensure stability.

The set of λh for which the growth factor is less than one is called the *linear stability domain* $\mathcal{D}$ (or *region of absolute stability*).

Example For Euler's method we have

$$|1 + \lambda h| < 1$$

so that (for complex λ)

$$\mathcal{D}_{Euler} = \{z = \lambda h \in \mathbb{C} : \ |z + 1| < 1\},$$

a rather small circular subset of the left half of the complex plane.

FIGURE

Example On the other hand, we can show that the implicit or *backward Euler* method

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$$

is *unconditionally stable* for the above problem.

To see this we have

$$y_{n+1} = y_n + h\lambda y_{n+1}$$

or

$$y_{n+1} = \frac{1}{1 - \lambda h} y_n.$$

Therefore,

$$y_n = \left(\frac{1}{1-\lambda h}\right)^n y_0.$$

Now, for (real) $\lambda < 0$, the growth factor

$$\left(\frac{1}{1-\lambda h}\right) < 1$$

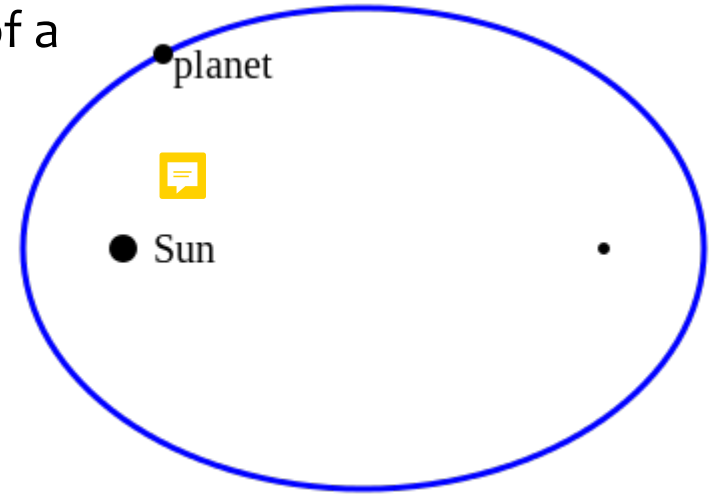
for any $h > 0$, and we can choose the step size h arbitrarily large.

Of course, this statement pertains only to the stability of the method. In order to achieve an appropriate accuracy, h still has to be chosen reasonably small. However, as we will see below, we do not have to worry about stepsize constraints imposed by the *stiffness* of the problem.

Practice Session #3

- `kepler.cpp`: solve the equation of motion of a point mass in a central gravitational field,

$$\frac{d^2\vec{x}}{dt^2} = -\frac{GM}{r^2}\hat{r}$$



Use dimensionless units by setting GM=1 and set the initial mass at the point (`x=4,y=0`) with initial velocity in the y-direction. Use two dimensions (x-y) only.

→ What is the maximum velocity for which the orbit is closed ?

→ Consider first the case of a circular orbit and integrate for 10 orbits by counting turning points (1 orbit = 2 turning points). How can we safely choose the step size ?

→ Now consider an elliptical orbit. Can you devise a strategy to control the time step so that $\Delta\theta \approx$ const ? (Make sure your algorithm produces bounded orbits…)

→ How would you scale your results to physical c.g.s units ?

Useful Tips (& tricks)

- Bounded orbits are admissible if the mechanical energy of the system is negative, i.e., $E = v^2/2 - 1/r < 0 \rightarrow v < \mathrm{sqrt}(2/r)$

- Circular orbits are defined by matching centripetal and gravitational forces: $v^2/r = 1/r^2 \rightarrow v = \mathrm{sqrt}(1/r)$

- In general one could write $v = \mathrm{sqrt}(\alpha/r)$ and, for $\alpha < 2$, we always have *elliptical orbits* which repeats.

- In 2D we have a total of 4 ODE,

$$\begin{cases} \dfrac{dx}{dt} = v_x \\ \dfrac{dy}{dt} = v_y \\ \dfrac{dv_x}{dt} = -\dfrac{x}{r^3} \\ \dfrac{dv_y}{dt} = -\dfrac{y}{r^3} \end{cases}$$

where $r = \mathrm{sqrt}(x^2 + y^2)$

Scaling the Equations to Physical Units

- By properly choosing the dimensions, one can easily scale the equations to dimensionless form.

- Write a physical quantity q (in cgs) as $q = q_{cgs}\ q_c$ where q_{cgs} is a constant in cgs units representing your reference unit and q_c is a dimensionless (code) variable.
- For instance, by writing time $t = t_{cgs}\ t_c$, and $r = L_{cgs}\ L_c$, one has

$$\begin{aligned} \frac{d\vec{x}}{dt} &= \vec{v} \\ \frac{d\vec{v}}{dt} &= -\frac{GM}{r^2}\hat{r} \end{aligned} \quad \Longrightarrow \quad \begin{cases} \vec{x} \rightarrow x_{cgs}\vec{x}_c \\ \vec{v} \rightarrow (L_{cgs}/t_{cgs})\vec{v}_c \end{cases} \quad \Longrightarrow \quad \begin{aligned} \frac{d\vec{x}_c}{dt_c} &= \vec{v}_c \\ \frac{d\vec{v}_c}{dt_c} &= -\frac{t_{cgs}^2}{L_{cgs}^3}\frac{GM}{r_c^2}\hat{r} \end{aligned}$$

- Choosing $t_{cgs}^2 = L_{cgs}^3/GM$ and keeping L_{cgs} arbitrary we obtain the dimensionless form of the equations.

- Make sure you choose independent reference units (do not choose length, time and velocity !!).

Symplectic Integrators

- Many problems require integrating Newton's equations of motion over a long period of time and for a large number of particles.

- In the case of a large number of interacting particles, we need to compute accelerations on every particle. It can be challenging to use a variable step size integrator, so more commonly a low order integrator is used on all particles during each time-step. The step size can be chosen so that it is appropriate for the particle with the largest acceleration.

- Symplectic integrators are designed to preserve the geometry of phase space and a Hamiltonian system is integrated that is an approximation to the desired Hamiltonian.

- Errors are bounded and the integration exhibits stability on long timescales.

Symplectic Integrators

- Both algorithms take advantage of the property that the equation for dx/dt does not involve x itself and the equation for dv/dt does not involve v (assuming velocity independent forces).

$$\frac{dx}{dt} = v$$
$$\frac{dv}{dt} = \frac{F(x)}{m} = a(x)$$

- The Leapfrog and Verlet algorithms are particularly suited for this purpose and they possess the following properties:

  - These integrators are *time-reversible*: one can integrate forward n steps and then reverse the direction to arrive at the same starting position.
  - They also *conserves the* (slightly modified) *energy* of dynamical systems, which can be especially useful in orbital dynamics [many other integration schemes, such as the (order-4) Runge-Kutta method, do not conserve energy and allow the system to drift substantially over time].
  - In a spherically symmetric potential, angular momentum is conserved and, remarkably, the leapfrog/(velocity or position) Verlet algorithm conserves it exactly.

Leapfrog Method

- Consider the equation of motion → assuming that the force does not depend on velocity.

$$\frac{dx}{dt} = v$$
$$\frac{dv}{dt} = \frac{F(x)}{m} = a(x)$$

- The Euler method would approximate this equation using $x^{n+1} = x^n + hv^n$

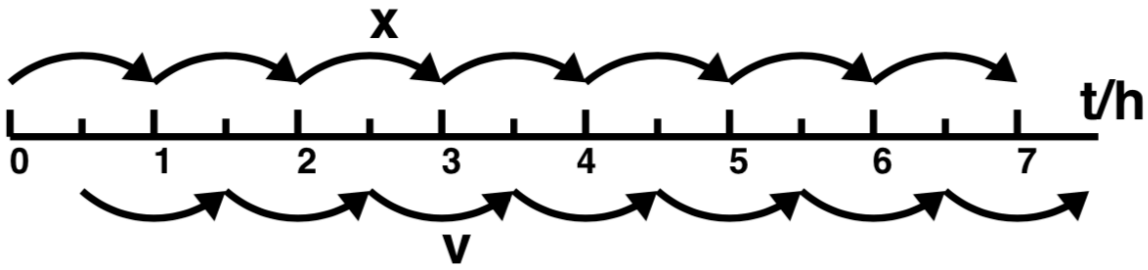
- A better approximation would be to replace the v^n by its value at the midpoint of the interval:

$$x^{n+1} = x^n + hv^{n+\frac{1}{2}}$$

- If $v^{n+1/2}$ is known, then we can apply a similar trick to evolve v forward in time:

$$v^{n+\frac{3}{2}} = v^{n+\frac{1}{2}} + ha(x^{n+1})$$

- Thus, once we have started off with x^o and $v^{1/2}$ we can continue with x and v leapfrogging over each other as shown in the figure below:



Leapfrog Method

- The basic integration formula for the leapfrog algorithm is therefore:

$$\boxed{x_{n+1} = x_n + h v_{n+1/2}\,,}$$

$$\boxed{v_{n+3/2} = v_{n+1/2} + hF(x_{n+1})}$$

- This method has a local error ≈ h^3 and therefore is a second-order accurate method.

- This method, although symplectic, has one major drawback: it requires position and velocity to be staggered (displaced by h/2 in time).

- It also requires a (strictly) constant h, in order to preserve second-order accuracy in time.

The velocity-Verlet Method

- One way to overcome the limitations of the Leap-frog method is to average the velocity, $v^n = (v^{n-1/2}+v^{n+1/2})/2$ and

$$\begin{aligned} v^{n+\frac{1}{2}} &= v^n + \frac{h}{2}a(x^n) \\ x^{n+1} &= x^n + hv^{n+\frac{1}{2}} \\ v^{n+1} &= v^{n+\frac{1}{2}} + \frac{h}{2}a(x^{n+1}) \end{aligned}$$

- This is called the *velocity Verlet* algorithm. It is equivalent to the Leapfrog scheme with an additional recipe for starting the algorithm off and for evaluating v and x at the same time.

- Apparently, it seems that two force calculations are required per time step, but a closer look reveals that the acceleration term in the 3rd line is the same as the force in the 1st line of the next step, so it can be stored and reused.

- Since velocity Verlet is the same as leapfrog, it is a second order method.

The position-Verlet Algorithm

- Note that instead of starting with a half step for v followed by full step for x and another half step for v, one could do the opposite: a half step for x followed by full step for v and another half step for x, i.e.

$$\begin{aligned} x^{n+\frac{1}{2}} &= x^n + \frac{h}{2}v^n \\ v^{n+1} &= v^n + ha(x^{n+\frac{1}{2}}) \\ x^{n+1} &= x^{n+\frac{1}{2}} + \frac{h}{2}v^{n+1} \end{aligned}$$

- This is called the *position Verlet* algorithm.

Generalization to Systems

- Both the velocity-Verlet and the position-Verlet (as well as the leap-frog method) can be easily generalized to the case of many equations.

- For instance, for the velocity-Verlet, one has

$$
\begin{aligned}
v_i^{n+\frac{1}{2}} &= v_i^n + \frac{h}{2} a_i(x^n) && (i = 1, ..., N) \\
x_i^{n+1} &= x_i^n + h v_i^{n+\frac{1}{2}} && (i = 1, ..., N) \\
v_i^{n+1} &= v_i^{n+\frac{1}{2}} + \frac{h}{2} a_i(x^{n+1}) && (i = 1, ..., N)
\end{aligned}
$$

- Where $\mathbf{x}_i$ and $\mathbf{v}_i$ are the positions and velocities, $\mathbf{a}_i$ is the acceleration for the corresponding component.
- Note that each acceleration depends, of course on the set of all position $\{\mathbf{x}_i\}$

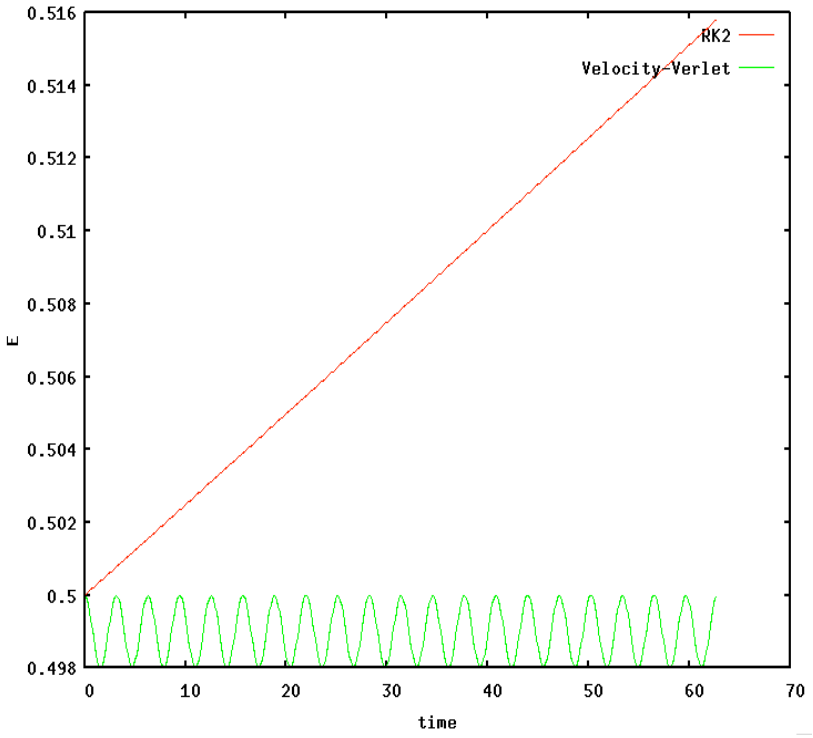
- It is important that all the positions are first updated, then all the forces are calculated using the new positions, then all the velocities are updated and finally all the positions are updated again.

Practice Session #4

- `harmonic.cpp`: solve the harmonic oscillator problem

$$\frac{d^2x}{dt^2} = -\omega^2 x$$

using velocity-Verlet (or position-Verlet) algorithm, choosing *a constant time step* `h = 0.02T` and initial condition $x_0 = 1$, $v_0 = 0$;



→ Evolve the system for 10 periods and compute the energy of the system. Compare it with the RK2 midpoint.

A fourth-order Symplectic Algorithm

- The simplest higher order symplectic algorithm is that of E. Forest and R.D. Ruth:

$$
\begin{aligned}
v &\leftarrow v + \gamma \frac{h}{2} a(x) \\
x &\leftarrow x + \gamma h\, v \\
\\
v &\leftarrow v + (1-\gamma) \frac{h}{2} a(x) \\
x &\leftarrow x + (1-2\gamma) h\, v \\
\\
v &\leftarrow v + (1-\gamma) \frac{h}{2} a(x) \\
x &\leftarrow x + \gamma h\, v \\
\\
v &\leftarrow v + \gamma \frac{h}{2} a(x)
\end{aligned}
$$

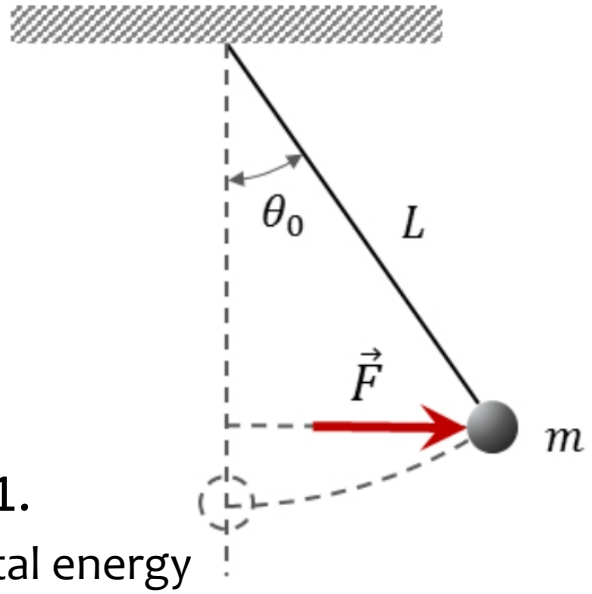
where $\gamma = \dfrac{1}{2 - 2^{1/3}}$

Note that this method requires 3 evaluations of the force per time step, as opposed to just one for the leapfrog method.
Note too that the steps are symmetric about the middle one (this ensures time reversal invariance) .

Practice Session #5

- `pendulum.cpp`: solve the equation for the (simple gravity) pendulum

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin\theta$$



with initial condition $\theta_0 = \pi/4$, $d\theta/dt|_0 = 0$.

Use $\Delta t = h = 0.1$ and integrate for 50 periods.

Make the problem dimensionless and set $g/L = 1$.

→ Compute the (absolute value of) relative error of the total energy

$$\frac{E}{m} = \frac{1}{2}v^2 + gL(1-\cos\theta) \quad \rightarrow \quad \frac{E}{mL^2} = \frac{\omega^2}{2} + \frac{g}{L}(1-\cos\theta)$$

using RK2 and Verlet-Velocity (or also RK4 and optionally Forest-Ruth).

Discuss.

→ Use a smaller initial amplitude ($\theta_0 = \pi/40$) and verify your results with the classical solution in the small angle limit ($\sin(\theta) \approx \theta$).

→ How can you scale to c.g.s units ?