Ordinary Differential Equations

Lecture 7, Computational Astrophysics (ASTR660)

Hsiang-Yi Karen Yang, NTHU, 10/27/2022

Announcements

- ***HW3 is due today.*** Late submission within one week will receive ***75%*** of the credit
- Schedule for the midterm presentation has been posted on eLearn. Please let me know ASAP if you have any question. ***Please put the title of your talk in the following spreadsheet:*** https://docs.google.com/spreadsheets/d/16zh4Xw3bOGANhSBlbKo6is6DRghPC3Nul8tmeOVIl8c/edit?usp=sharing
- ***Evaluation sheet*** for the midterm presentation has also been posted on eLearn. Please prepare your presentation accordingly.

Date	Time	Name	Title
11/3/22	14:25-14:40	李佳倫	
	14:40-14:55	石郡翰	
	14:55-15:10	凌志騰	
	15:30-15:45	張修瑜	
	15:45-16:00	簡嘉成	
	16:00-16:15	龔一桓	
11/10/22	14:25-14:40	劉一璠	
	14:40-14:55	郭弈翔	
	14:55-15:10	胡英祈	
	15:30-15:45	周育如	
	15:45-16:00	吳耕緯	
	16:00-16:15	鄭文淇	
	16:30-16:45	謝明學	
	16:45-17:00	考司圖巴	

Evaluation sheet for the midterm presentations

Name of Presenter:

Title of Presentation:

	Poor				Excellent
CONTENTS OF PROJECT PROPOSAL	**1**	**2**	**3**	**4**	**5**
Were the scientific motivations of the project clearly described?	,	,	,	,	,
Was proper background information on the topic given?	,	,	,	,	,
Was the proposed method suitable for the scientific problem to be tackled?	,	,	,	,	,
Was the numerical method chosen for this project clearly explained?	,	,	,	,	,
Was the motivation for choosing the numerical method clearly described?	,	,	,	,	,
Did the presenter provide sufficient justification for the feasibility of the project? ...	,	,	,	,	,
Did the presenter have a clear understanding of the material presented?	,	,	,	,	,
PRESENTATION SKILLS	**1**	**2**	**3**	**4**	**5**
Were the main ideas presented in an orderly and clear manner?	,	,	,	,	,
Did the presentation fill the time allotted?	,	,	,	,	,
Were the visual aids well prepared with clear and understandable figures/text?	,	,	,	,	,
Did the presentation appropriately cite relevant references?	,	,	,	,	,
Did the speaker make good eye contact and maintain the interest of the audience? ...	,	,	,	,	,
How well did the presenter handle questions from the audience?	,	,	,	,	,

COMMENTS/ENCOURAGEMENTS/CONSTRUCTIVE CRITICISMS/WHAT DO YOU LIKE ABOUT THIS PRESENTATION?

Previous lecture – linear systems $\boldsymbol{A \cdot x = b}$

- In physics/astrophysics, we often need to solve equations that are *linear* in the unknown variables
- Express ***A*** as a *M* x *N* matrix, # solutions depend on values of *M* and *N*, and whether ***A*** is singular or non-singular
- *Vector norms* (*L1*-norm, *L2*-norm, *L*∞-norm) can measure the magnitude of a vector and often are used for quantifying errors

  Matrix norms can measure magnitude of a matrix
- *Condition number* is used to see how *sensitive* the solutions are

$$\text{cond} = \frac{|[f(\hat{x}) - f(x)]/f(x)|}{|(\hat{x} - x)/x|} = \frac{|\Delta y/y|}{|\Delta x/x|}$$

$$\text{cond}(A) = ||A|| \cdot ||A^{-1}||$$

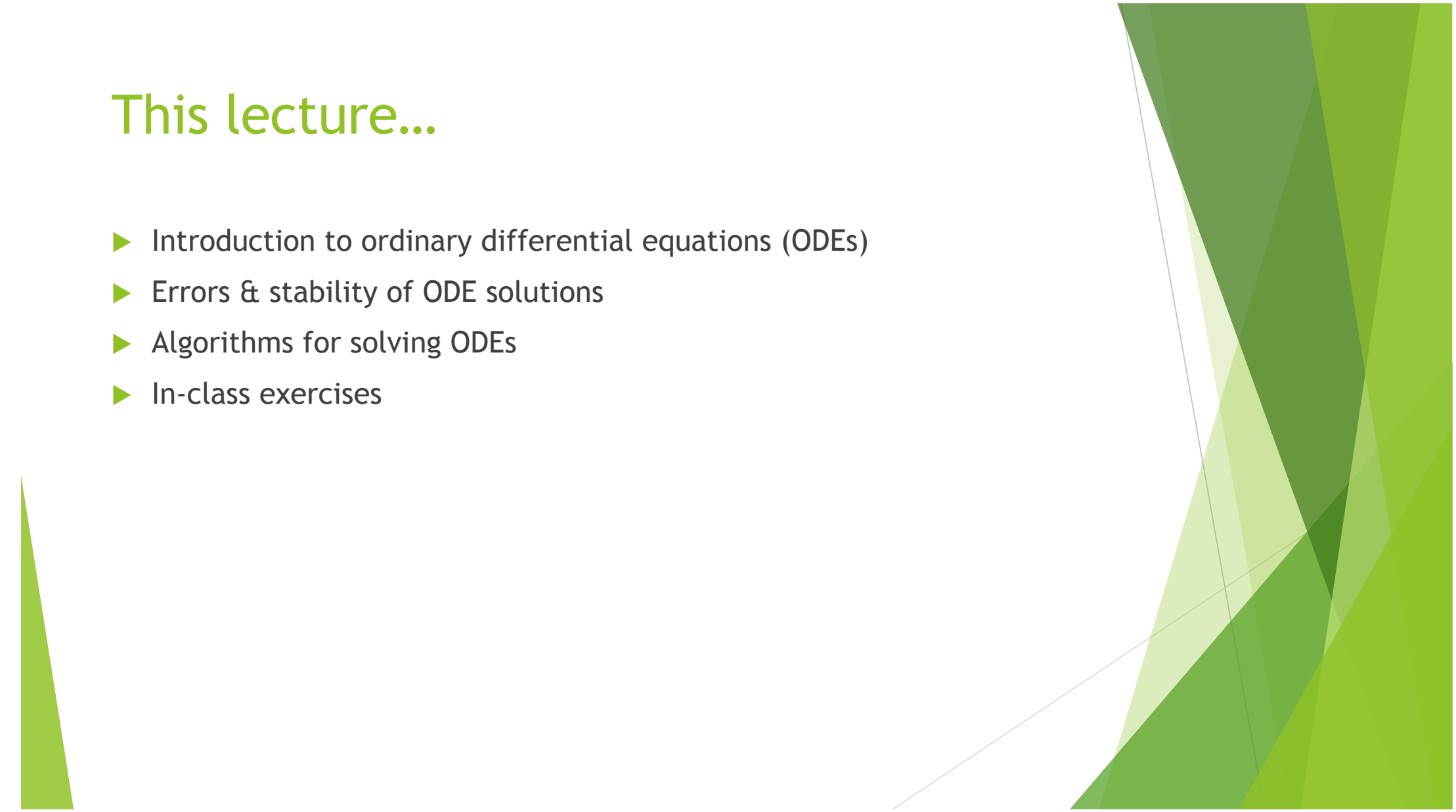
Previous lecture -- linear systems

$$A \cdot x = b$$

- Strategies for solving linear systems
  - Transform a general matrix to a triangular matrix: *Gaussian elimination* or *LU decomposition/factorization*
  - For lower triangular systems: *forward substitution*
  - For upper triangular systems: *back substitution*
- For research purposes, it is recommended to use industrial-strength matrix software

This lecture…

- Introduction to ordinary differential equations (ODEs)
- Errors & stability of ODE solutions
- Algorithms for solving ODEs
- In-class exercises



Intro to ordinary differential equations

Ordinary differential equations (ODEs)

- Differential equations involve derivatives of unknown solution functions
- *Ordinary differential equations*: all derivatives are w.r.t *single independent variable* (e.g., time)
- *Order* is determined by the highest-order derivative, e.g., *k*-th order ODE:

$$y^{(k)}(t) = f\left(t, y, y', \ldots, y^{(k-1)}\right)$$

- Solution functions (y, y', $y^{(k-1)}$) are continuous in *t*, but the *numerical* solutions are *approximate* solutions at *finite* points in *t*

Ordinary differential equations (ODEs)

- ODE with higher-order derivatives can be transformed into first-order systems
- **Example:**

$$y''' = f(t) \quad \rightarrow \quad \begin{bmatrix} y' = y_1 \\ y_1' = y_2 \\ y_2' = f(t) \end{bmatrix}$$

$$F = ma = mx'' \quad \rightarrow \quad \begin{bmatrix} x' = v \\ v' = a = F/m \end{bmatrix}$$

Ordinary differential equations (ODEs)

- General 1st-order system of ODEs has the form:

$$\boldsymbol{y}'(t) = \boldsymbol{f}(t, \boldsymbol{y}) \qquad \begin{bmatrix} y_1'(t) \\ y_2'(t) \\ \dots \\ y_n'(t) \end{bmatrix} = \begin{bmatrix} dy_1(t)/dt \\ dy_2(t)/dt \\ \dots \\ dy_n(t)/dt \end{bmatrix} = \begin{bmatrix} f_1(t, \boldsymbol{y}) \\ f_2(t, \boldsymbol{y}) \\ \dots \\ f_n(t, \boldsymbol{y}) \end{bmatrix}$$

- Function $\boldsymbol{f}$ is given and we wish to find unknown solution functions $y_1(t)$, $y_1(t)\dots$ $y_n(t)$
- We can use numerical methods for *1st-order ODEs* for solving this system

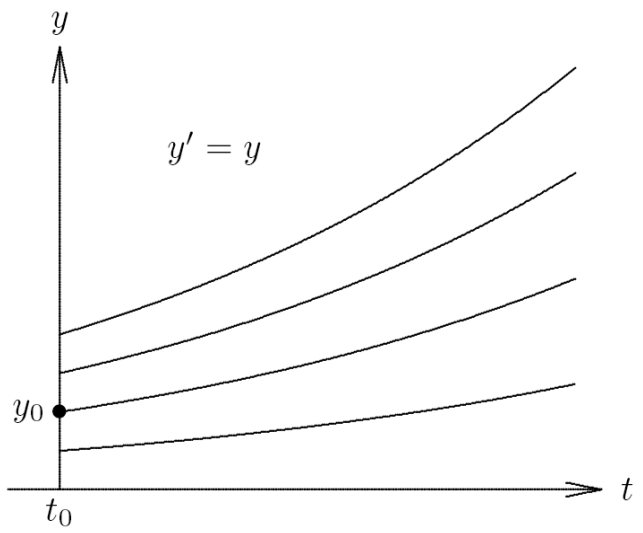
Initial value problems

- The ODE $\boldsymbol{y}'(t) = \boldsymbol{f}(t, \boldsymbol{y})$ does not determine a unique solution function because only the derivatives (slopes) are specified
- If $\boldsymbol{y}(t)$ is solution and $\boldsymbol{c}$ is any constant, then $\boldsymbol{y}(t) + \boldsymbol{c}$ is also a solution
- Thus, an initial value is required to determine a unique solution, e.g., $\boldsymbol{y}(t_0) = \boldsymbol{y}_0$
- That is why we often call this ***"initial value problems (IVPs)"***

Initial value problems -- example

- Consider a 1st-order ODE: $y' = y$

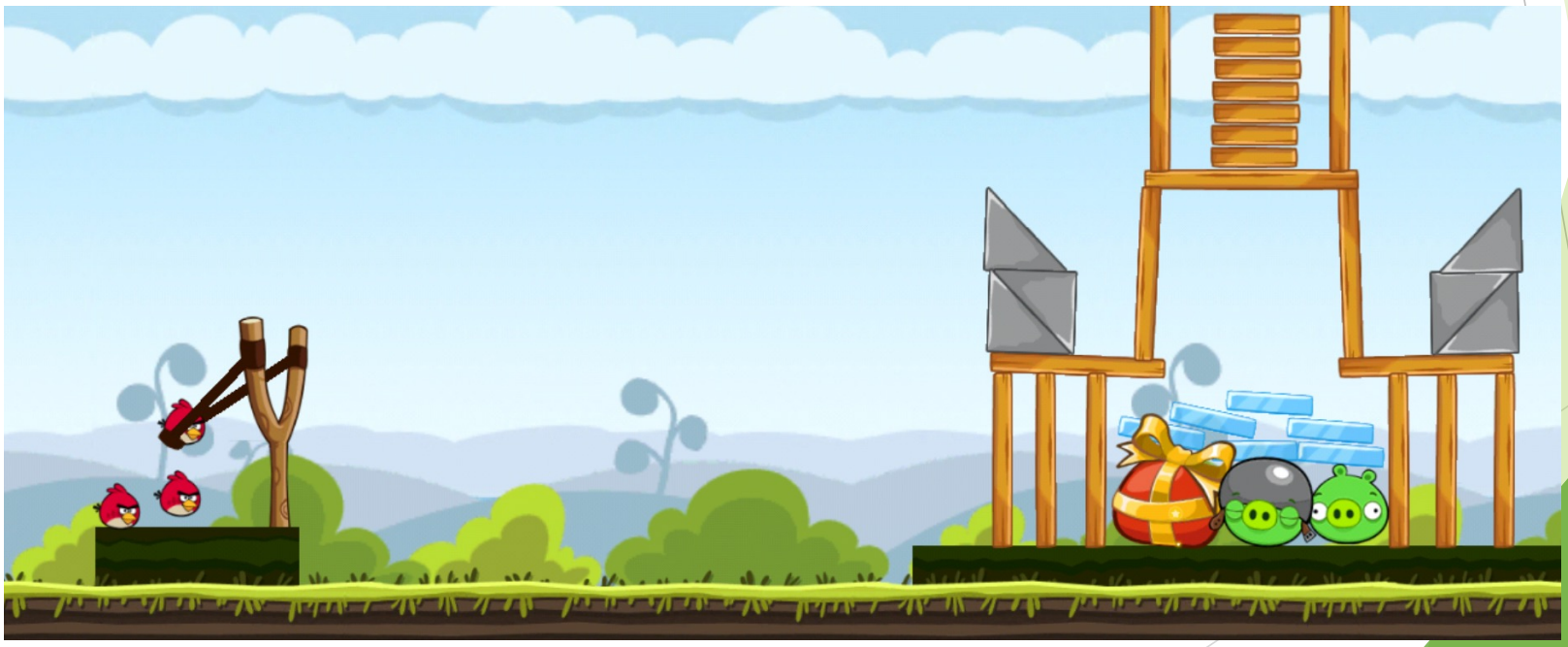
- Family of solutions if given by $y(t) = ce^t$, where c is a constant
- Imposing initial condition $y(t_0) = y_0$ determines a unique solution
- For example if $t_0 = 0$ then $c = y_0$ => $y(t) = y_0 e^t$



ODEs – example #1

- Simulating the ***trajectories*** of the angry bird (or a missile) given the initial position and initial velocity

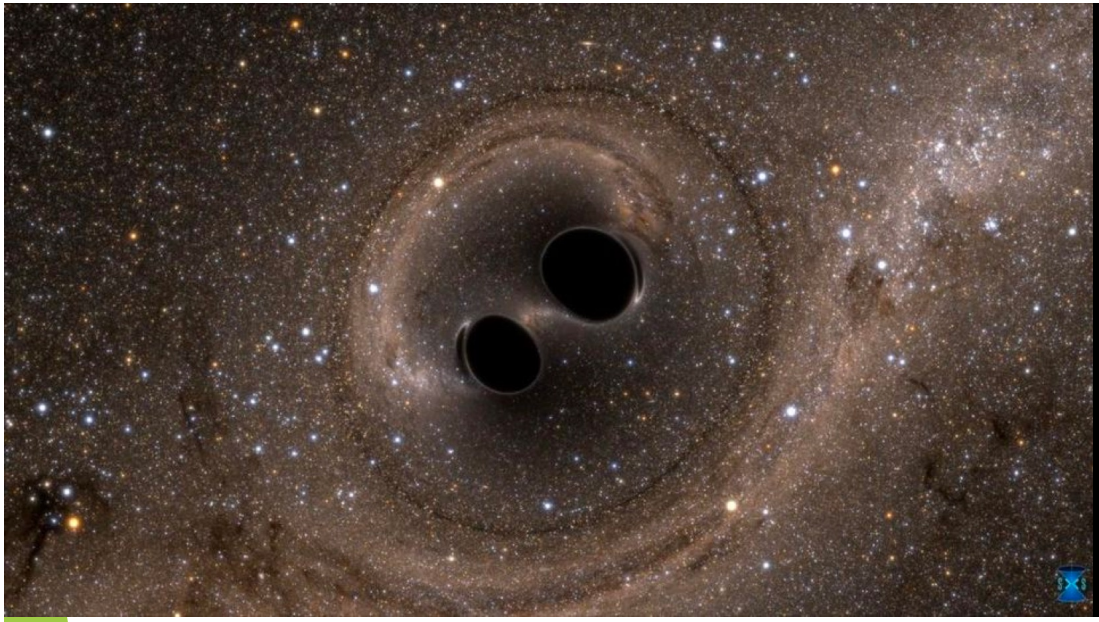
$$\ddot{x} = a_x = 0$$
$$\ddot{y} = a_y = -g$$

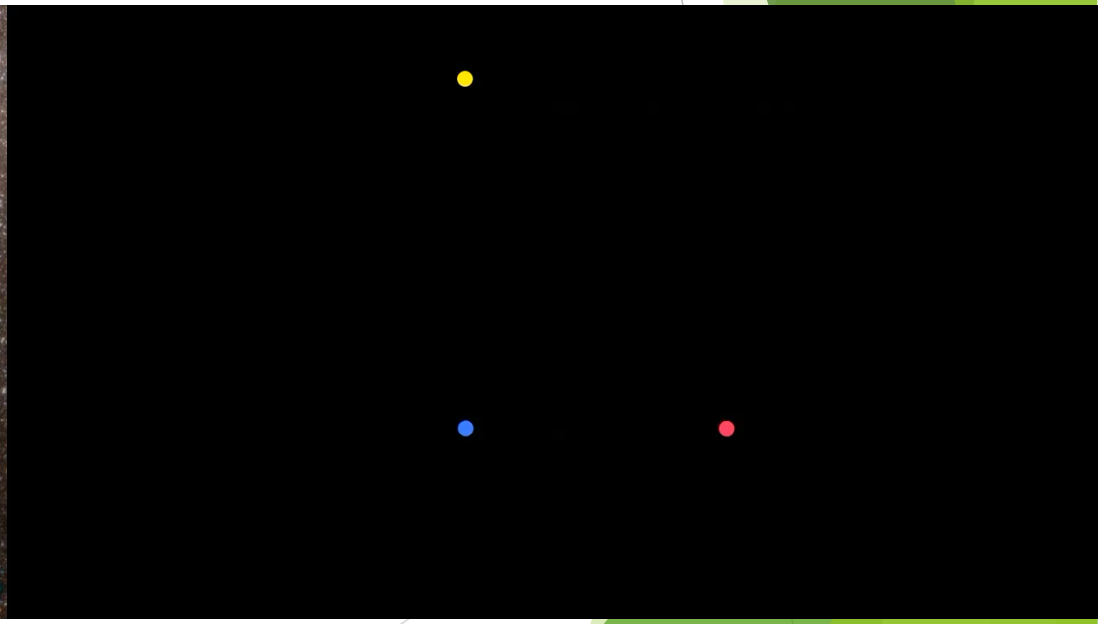


ODEs – example #2

- ***N-body simulations*** of the orbits of celestial objects

$$\boldsymbol{f} = \frac{GMm}{r^2}\hat{f} = m\ddot{\boldsymbol{x}}$$





Video link: https://youtu.be/cev3g826ilQ

Errors & stability of ODE solutions

Numerical solutions of ODEs $\boldsymbol{y}'(t) = \boldsymbol{f}(t, \boldsymbol{y})$

- Numerical solution of ODE is ***approximate*** values of the solution function at ***discrete*** set of points
- Starting from the ***initial value*** $\boldsymbol{y}_0(t_0)$, we evaluate the slope $\boldsymbol{f}(t_0, \boldsymbol{y}_0)$ to predict value $\boldsymbol{y}_1$ at $t_1 = t_0 + h$ for some ***suitably chosen step h***
- Step by step we can generate the approximate solution values

Algorithm #1: Euler's method $y'(t) = f(t, y)$

- Consider Taylor series:

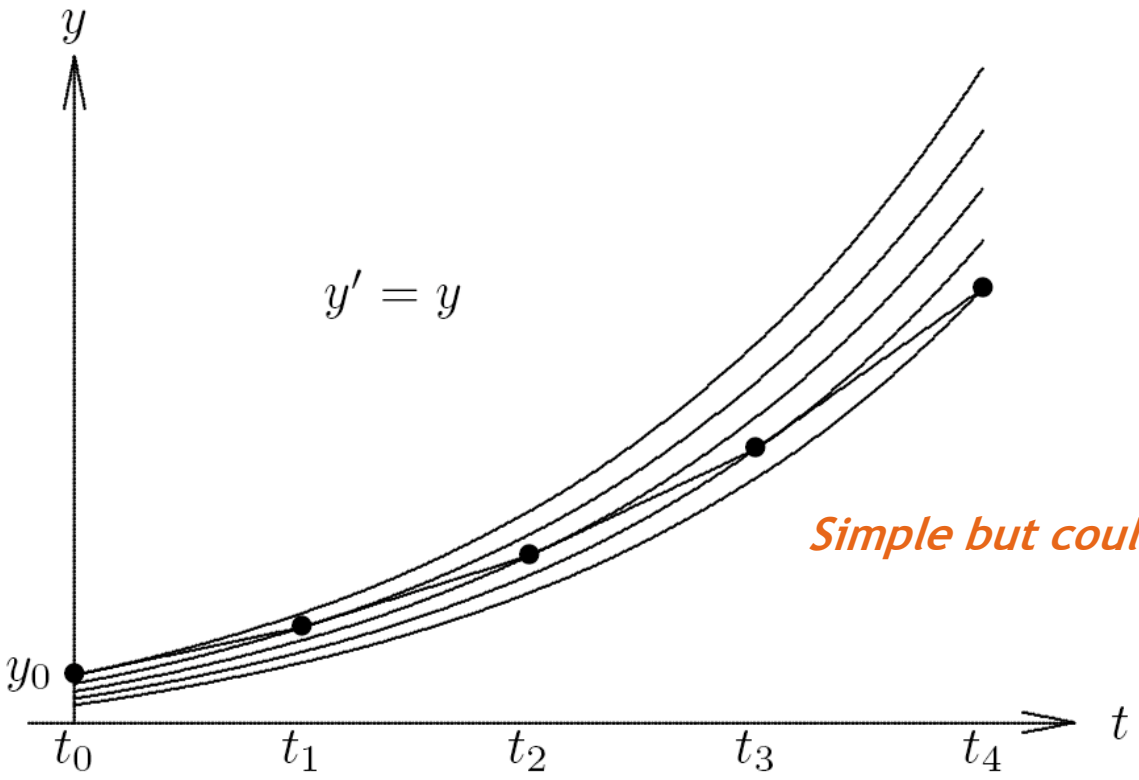
$$y(t+h) = y(t) + y'(t)h + \frac{y''(t)}{2}h^2 + \frac{y'''(t)}{6}h^3 + \ldots$$

- ***Euler's method*** -- consider only the 1st-order term to obtain the next approximate solution value:

$$y_{k+1} = y_k + h_k f(t_k, y_k)$$

- It advances solution by extrapolating along straight lines whose slope is given by $f(t_k, y_k)$
- It only requires info at one point in time to advance to next point

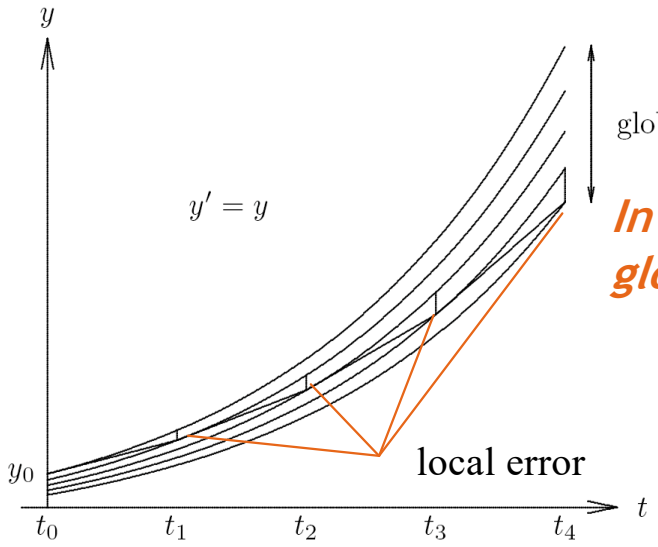
Euler's method -- example



Simple but could be very wrong!!

Errors of ODE solutions

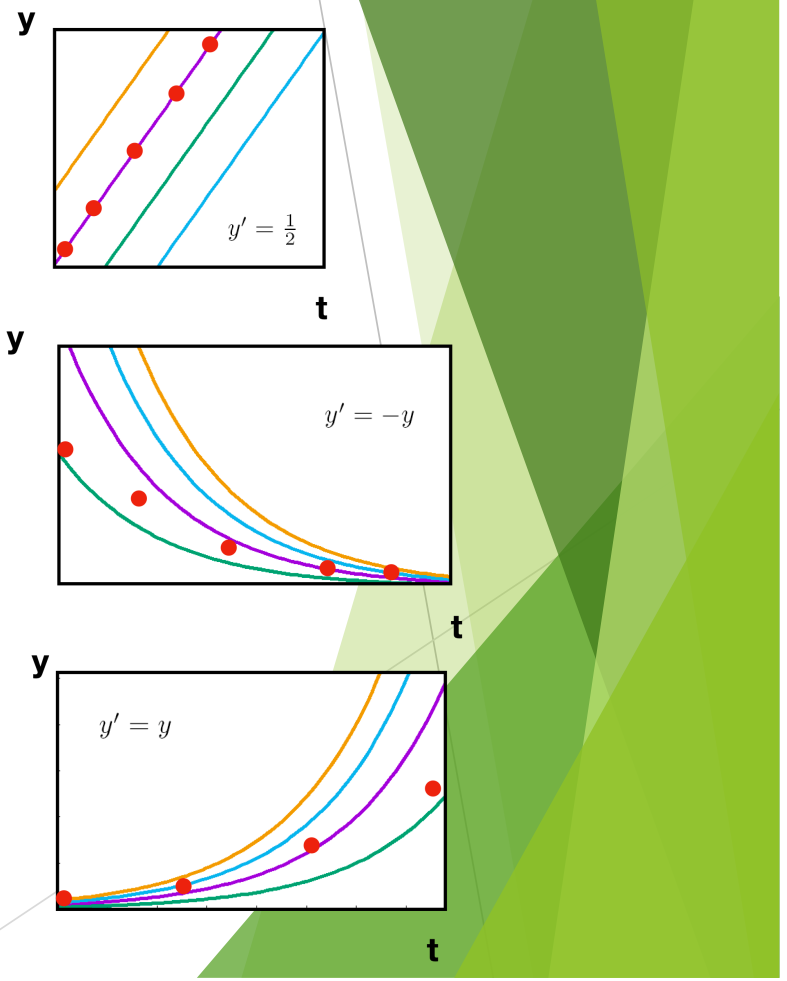
- The dominant error of ODE solutions is the ***truncation error***
- ***Local error***: error made in one step due to accuracy of numerical method
- ***Global error***: difference between computed solution and true solution
- To reduce global errors, we have to control local errors by choosing adequate numerical methods
- **Example:**



In this example, solution is "unstable" - global error grows with time

Stability of ODE solutions

- ***Stable***: if solutions resulting from perturbations of initial value remain close to original solution
- ***Asymptotically stable***: if solutions due to perturbations converge back to original solution
- ***Unstable***: if solutions due to perturbations diverge away from original solution without bound



Stability and accuracy of numerical method

- **Example**: solving $y' = \lambda y$ using Euler's method with fixed step size h

$$y_{k+1} = y_k + h\lambda y_k = (1 + h\lambda)y_k$$

$$y_k = (1 + h\lambda)^k y_0$$

- *For* $\lambda < 0$, exact solution -> 0 as t increases, so numerical solution converges only if

$$|1 + h\lambda| < 1 \qquad \text{or} \qquad h \leq -\frac{2}{\lambda}$$

Stability criterion for the Euler's method

- ***Growth factor*** $1 + h\lambda$ agrees with exact solution $e^{h\lambda} = 1 + h\lambda + \frac{(h\lambda)^2}{2} + \frac{(h\lambda)^3}{6} + \cdots$ through terms of 1st order in h, so ***Euler's method is 1st-order accurate***

Stability of ODE solutions

- In general, growth factor and thus the stability of ODE solutions depends on
  - numerical method, which determines the form of growth factor
  - properties of the ODE itself (i.e., whether it is sensitive or not)
  - step size *h*

Stiff ODEs

- ***Stiff ODEs*** are those systems where the step size is severely limited by the stability criterion
- This often occurs when
  - the solution damps over time very quickly (e.g., damping of oscillation of a very “stiff” string)
  - the solution oscillates rapidly
  - a system in which physical processes have very different timescales
- Euler’s method is extremely inefficient for solving stiff ODEs due to severe stability criterion, but other numerical methods (e.g., implicit methods) are needed

Key considerations involved in solving ODEs

- ***Accuracy*** -- choose a numerical method with desired accuracy to minimize local errors
- ***Stability*** – choose a step size *h* that obeys the stability criterion of the chosen method
- ***Cost*** – enlarge *h* as much as allowed by the stability criterion to save computational cost

Algorithms for solving ODEs

Explicit and implicit methods

$$y'(t) = f(t, y)$$

- (Forward) Euler's method is an ***explicit method (顯式法)*** - it uses info at time t_k to advance solution to time t_{k+1}
- Larger stability region can be obtained by using info at t_{k+1}, which makes the method ***implicit (隱式法)***
- **Example**: Backward Euler's method

$$y_{k+1} = y_k + h_k f(t_{k+1}, y_{k+1})$$

- This is a nonlinear equation in the unknown y_{k+1}, which can be solved using iterative methods like Newton's method
- Good initial guess can be obtained from explicit method or from solution at previous time step

Implicit methods

- Requires extra work/time for solving y_{k+1}
- But generally have much ***larger stability region*** than explicit methods
- Greatly relieves step size constraints so suitable for solving stiff ODEs
- **Example**: Backward Euler's method for ODE $y' = \lambda y$

$$y_{k+1} = y_k + h_k f(t_{k+1}, y_{k+1})$$

$$(1 - h\lambda) y_{k+1} = y_k$$

$$y_k = \left(\frac{1}{1 - h\lambda}\right)^k y_0 \qquad \left|\frac{1}{1 - h\lambda}\right| \leq 1$$

Solution is stable for any h when $\lambda < 0$ - "unconditionally stable"

- Backward Euler's method still has only 1st-order ***accuracy*** though

Higher-order methods #1

- Higher-order accuracy can be achieved by averaging forward and backward Euler's methods, i.e., ***implicit trapezoid method***:

$$y_{k+1} = y_k + h_k(f(t_k, y_k) + f(t_{k+1}, y_{k+1}))/2$$

- This method is also ***unconditionally stable***
- And it has ***2nd-order accuracy***

Higher-order methods #2

- *Taylor series method*:

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \frac{f'''(x)}{6}h^3 + \ldots$$

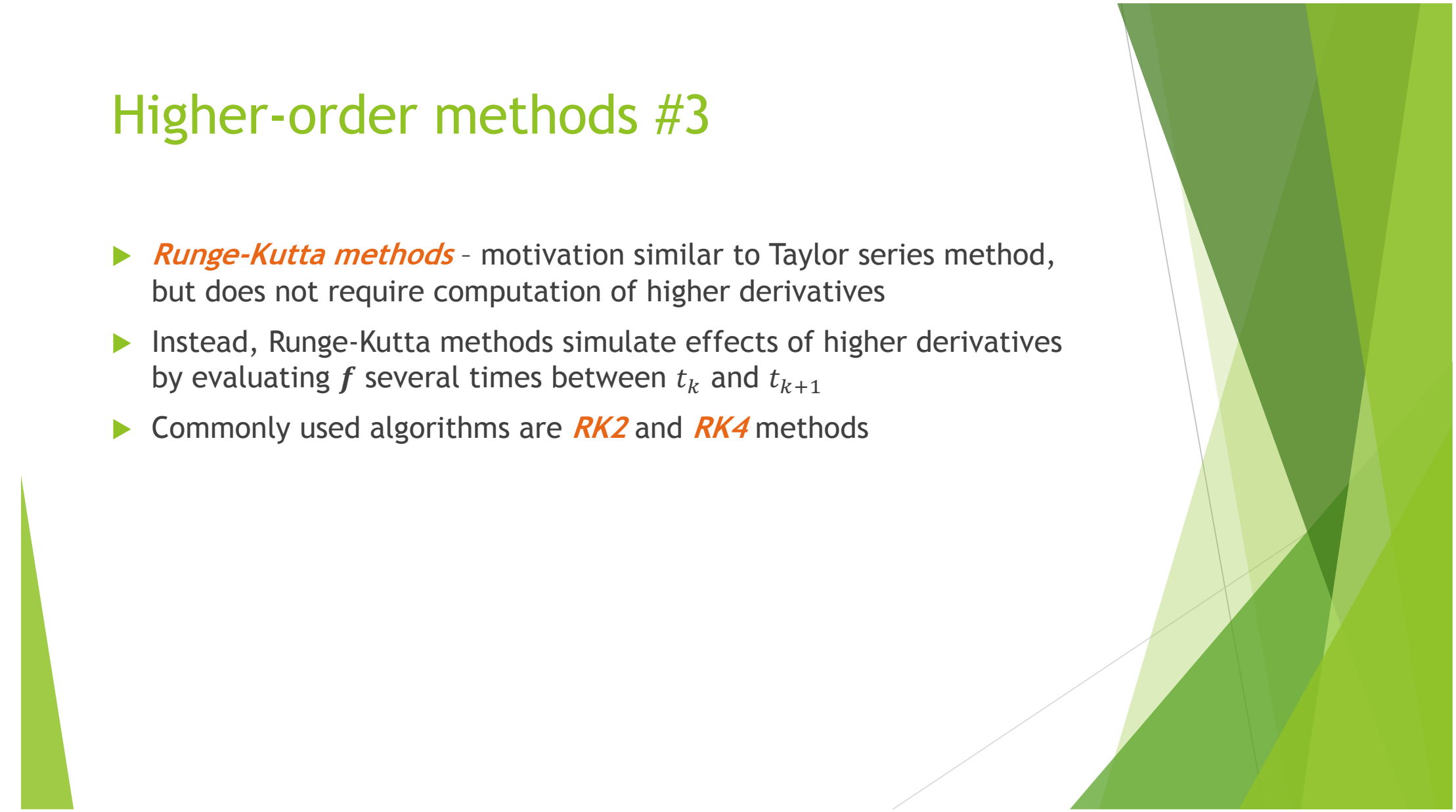
- Higher-order accuracy can be achieved by retaining more terms
- **Example:**

$$y_{k+1} = y_k + h_k y'_k + \frac{h_k^2}{2} y''_k$$

Difficult

Higher-order methods #3

- ***Runge-Kutta methods*** – motivation similar to Taylor series method, but does not require computation of higher derivatives
- Instead, Runge-Kutta methods simulate effects of higher derivatives by evaluating $\boldsymbol{f}$ several times between t_k and t_{k+1}
- Commonly used algorithms are ***RK2*** and ***RK4*** methods



RK2 (Runge-Kutta 2 or Heun's method)

$$y_{k+1} = y_k + \frac{h_k}{2}(k_1 + k_2)$$

$$k_1 = f(t_k, y_k)$$
$$k_2 = f(t_k + h_k, y_k + h_k k_1)$$

- Similar to implicit trapezoid method, but remains ***explicit***
- It has ***2nd-order accuracy***

RK4 (4th-order Runge-Kutta method)

$$y_{k+1} = y_k + \frac{h_k}{6}(k_1 + 2k_2 + 2k_3 + k4)$$

$$k_1 = f(t_k, y_k)$$
$$k_2 = f(t_k + h_k/2, y_k + (h_k/2)k_1)$$
$$k_3 = f(t_k + h_k/2, y_k + (h_k/2)k_2)$$
$$k_4 = f(t_k + h_k, y_k + h_k k_3)$$

- Analogous to Simpson's rule, but again remains ***explicit***

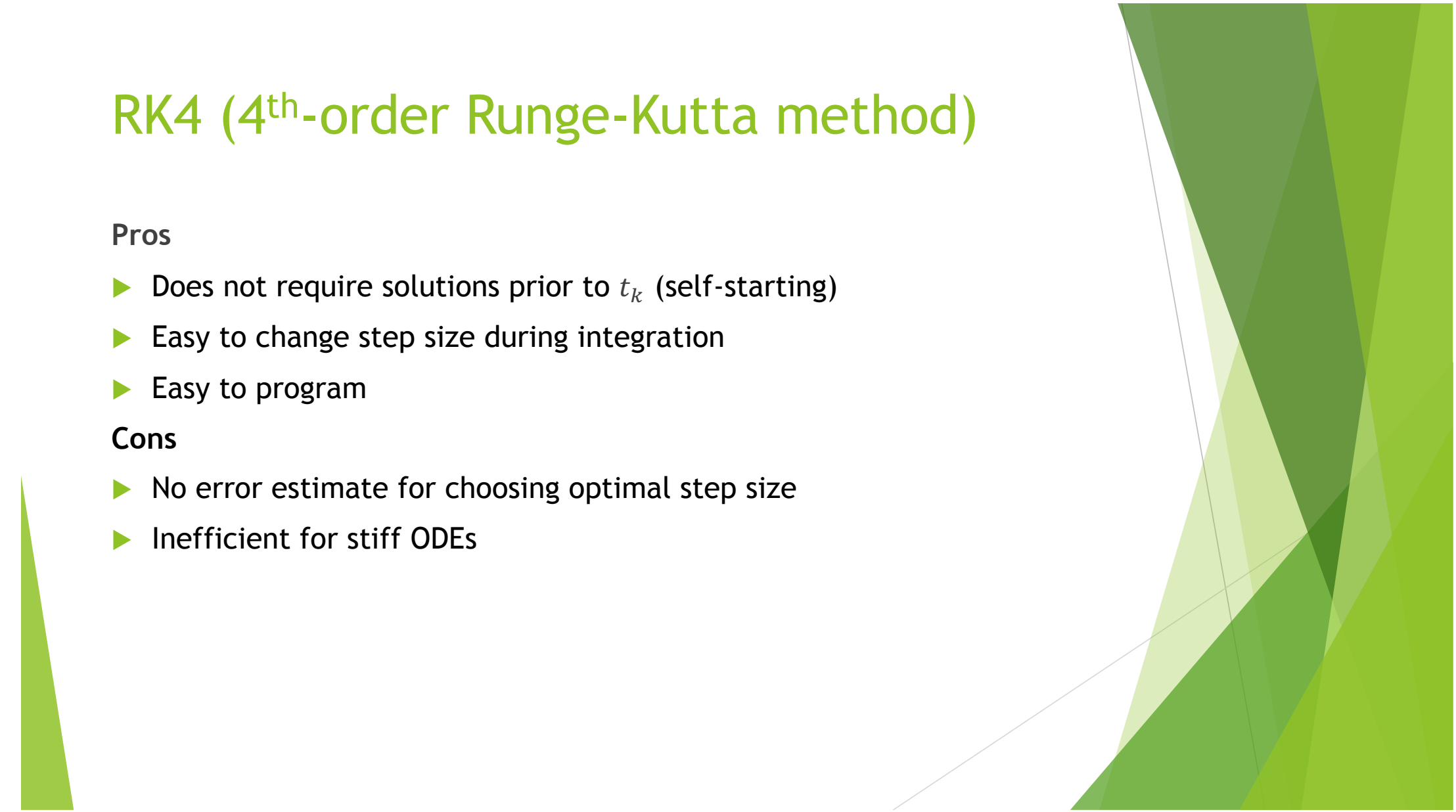
RK4 (4^{th}-order Runge-Kutta method)

Pros

- Does not require solutions prior to t_k (self-starting)
- Easy to change step size during integration
- Easy to program

Cons

- No error estimate for choosing optimal step size
- Inefficient for stiff ODEs

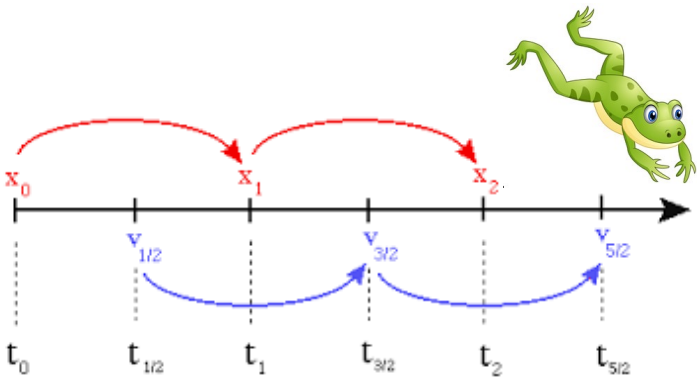


Higher-order methods #4

- ***Leapfrog method*** – algorithm designed for solving this type of ODE:

$$\dot{x} = v, \qquad \dot{v} = a$$

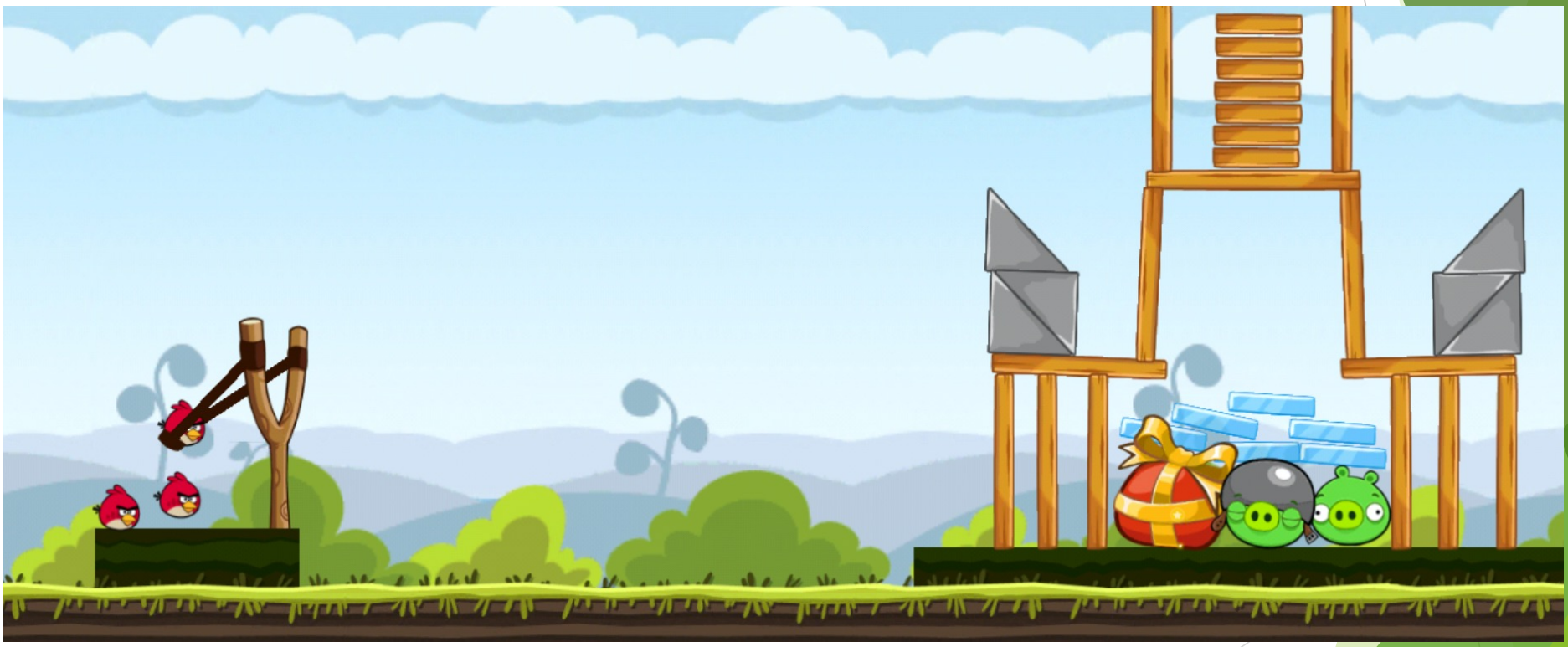
- Commonly used in physics/astrophysics for dynamical systems
- It is simple but has ***2nd-order accuracy***



$$v_{i+1/2} = v_{i-1/2} + a_i \, \Delta t,$$
$$x_{i+1} = x_i + v_{i+1/2} \, \Delta t,$$

In-class exercises

Exercise #1 – angry bird simulations



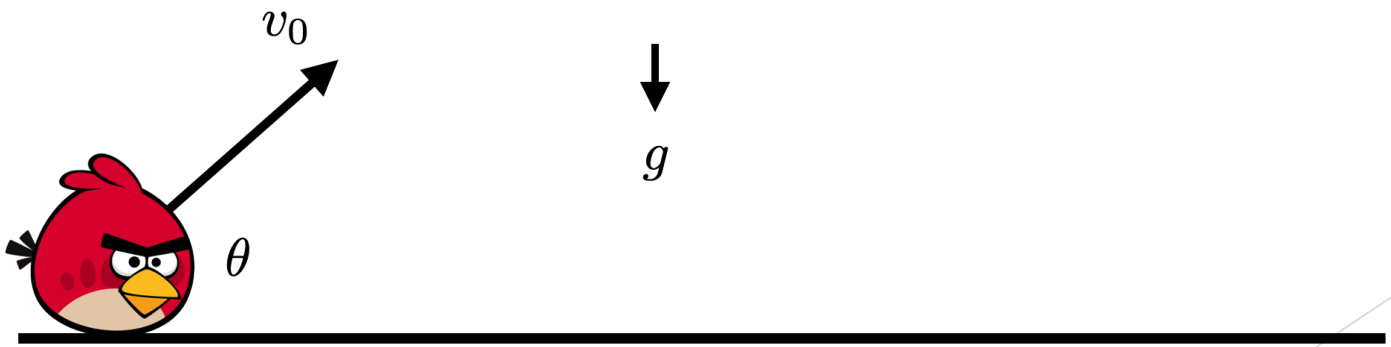
Exercise #1 – angry bird simulations

- First, convert the problem into four 1st-order ODEs

$$\ddot{x} = a_x = 0$$
$$\ddot{y} = a_y = -g$$

$$\dot{x} = v_x$$
$$\dot{y} = v_y$$
$$\dot{v_x} = a_x = 0$$
$$\dot{v_y} = a_y = -g$$

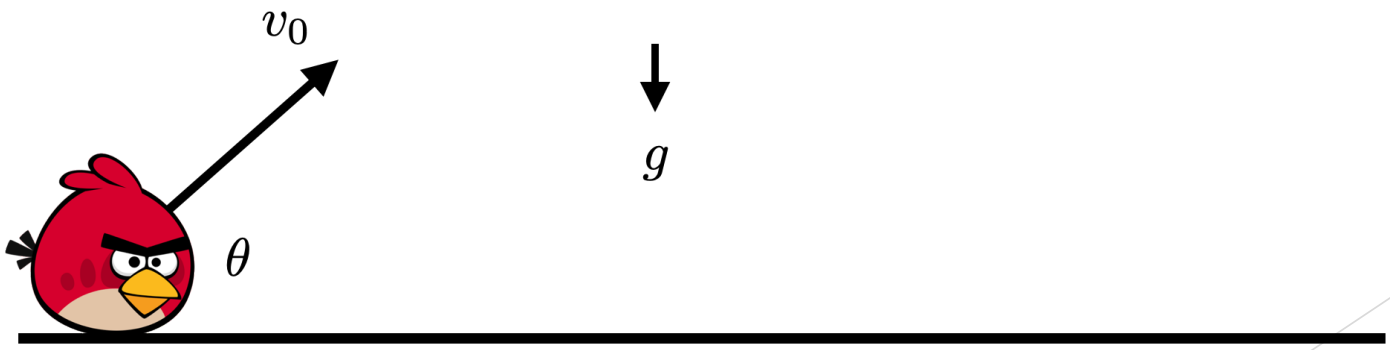
$$\boldsymbol{y}'(t) = \boldsymbol{f}(t, \boldsymbol{y})$$



Exercise #1 – angry bird simulations

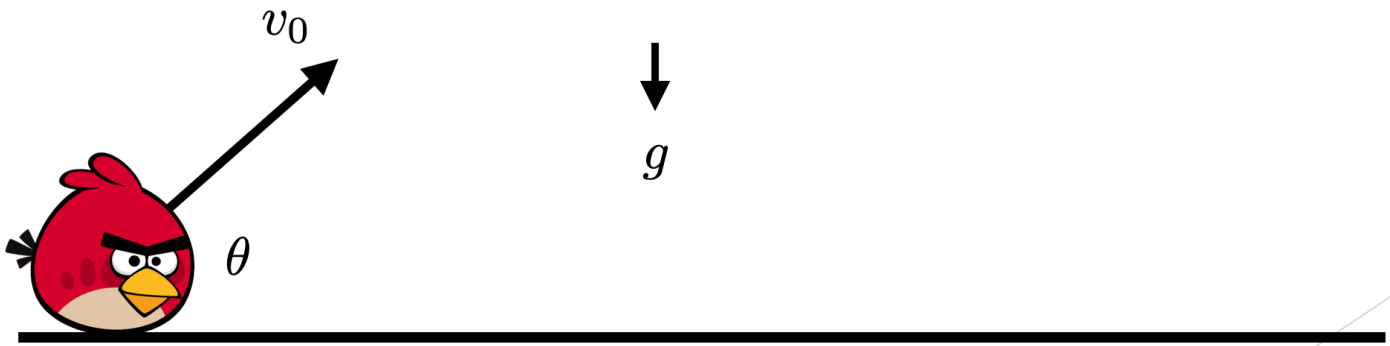
- Given the initial positions and velocities, use the Euler's method with a fixed time step Δt to advance the solution

$$
\begin{aligned}
x_{k+1} &= x_k + v_{x,k}\Delta t \\
y_{k+1} &= y_k + v_{y,k}\Delta t \\
v_{x,k+1} &= v_{x,k} + a_{x,k}\Delta t \\
v_{y,k+1} &= v_{y,k} + a_{y,k}\Delta t
\end{aligned}
$$



Exercise #1 – angry bird simulations

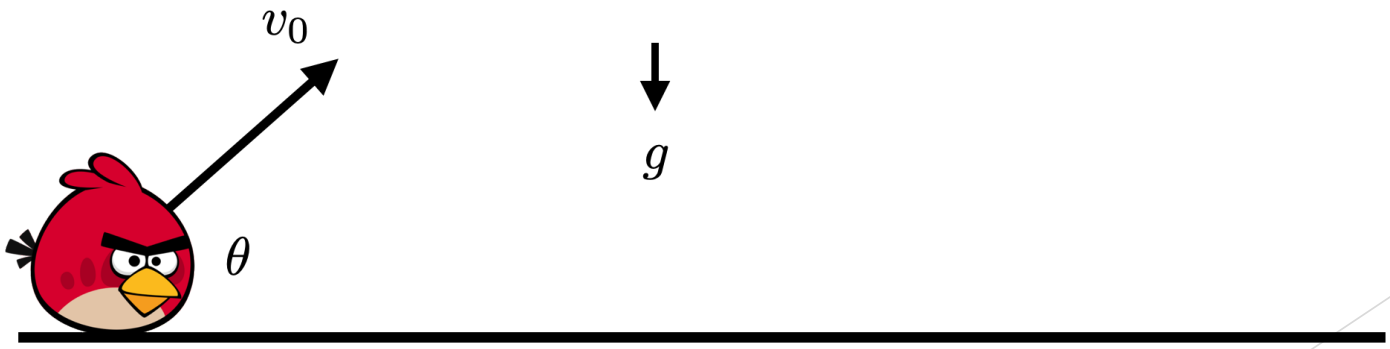
Initialize time, x0,y0, vx0, vy0, ax0, ay0
Do while (y >= 0)
    update x and y
    update vx and vy
    update ax and ay (if necessary)
    update time
    record trajectory
Enddo



Exercise #1 – angry bird simulations

- Compare with the analytical solution:

$$x = (v_0 \cos\theta)t$$
$$y = (v_0 \sin\theta)t - \frac{1}{2}gt^2$$



Exercise #1 – implement the Euler's method for simulating the angry bird trajectories

- Connect to CICA and load the Fortran compiler:

  ```
  module load pgi
  ```

- Go to your exercise directory on CICA:

  ```
  cd astr660/exercise
  ```

- Copy the template Fortran files to the current directory:

  ```
  cp -r /data/hyang/shared/astro660CompAstro/L7exercise ./
  ```

- Enter the angry_bird directory and see what files are there:

  ```
  cd L7exercise/angry_bird
  ls
  ```

angry.f90 – where the main program and initial conditions are set

```
program angry_bird

   use constants, only : show_constants, g ,pi, c
   use physics, only : update

   implicit none

   real :: angle, velocity

   real :: dt, time, velx, vely, posx, posy
   real :: anal_y, vy0

   !call show_constants()

   angle    = 60.0                ! degree
   angle    = angle * pi / 180.0  ! change to rad

   velocity = 30 ! m/s

   if (velocity .gt. c) then
       print *, "Error: the velocity cannot be faster than the speed of the light"
       stop
   endif

   velx = velocity * cos(angle)
   vely = velocity * sin(angle)

   dt   = 1.0  ! sec
   time = 0.0  ! initial time = 0.0

   posx = 0.0
   posy = 0.0

   anal_y = 0.0
   vy0    = vely

   print *, "# time,  posx,  posy,  velx,  vely, anal_y, err_y"
   print *, time, posx, posy, velx, vely, anal_y, 0.0

   do while (posy .ge. 0.0)

       call update(time, dt,posx,posy,velx,vely,posx,posy,velx,vely)
       time = time + dt
       anal_y = vy0*time - 0.5*g*time**2
       print *, time, posx, posy, velx, vely,anal_y,abs((anal_y-posy)/anal_y)
   end do

end program angry_bird
```

"only" use the subroutine "update" in the "physics" module

timestep used to advance the solution

while loop where the solutions are advanced

physics.f90 – where the module "physics" and the *ODE* is defined

Direct implementation of the Euler's method

Pack the variables into a subroutine so that we could change methods more easily

This defines the four ODE equations we are solving, i.e., y'(t)= f(t, y)

where the solvers (euler, rk2, rk4) are defined

```
module physics
   use solver
   implicit none
   contains
       subroutine update(time,dt,x0,y0,vx0,vy0,x,y,vx,vy)
           implicit none
           real, intent(in)  :: time, dt,x0, y0, vx0, vy0
           real, intent(out) :: x,y,vx,vy
           integer,parameter :: n = 4
           real,dimension(n) :: yin, ynext

           !x = x0 + vx0*dt
           !y = y0 + vy0*dt
           !vx = vx0 + 0.0*dt
           !vy = vy0 - g*dt

           ! pack yin
           yin(1) = x0
           yin(2) = y0
           yin(3) = vx0
           yin(4) = vy0

           call euler(4,yin,ynext,time,dt, my_func)
           !call rk2(4,yin,ynext,time,dt, my_func)
           !call rk4(4,yin,ynext,time,dt, my_func)

           ! unpack ynext
           x  = ynext(1)
           y  = ynext(2)
           vx = ynext(3)
           vy = ynext(4)

           return
       end subroutine update

       subroutine my_func(n,t,yin,k)
           use constants, only : g
           implicit none
           integer, intent(in)  :: n   ! number of ODEs
           real, intent(in)     :: t   ! not used here because dydt is constant
           real,dimension(n),intent(in)  :: yin   ! y
           real,dimension(n),intent(out) :: k     ! dydt

           ! n = 4 for the angry bird problem
           k(1)  =   ! vx
           k(2)  =   ! vy
           k(3)  =   ! ax
           k(4)  =   ! ay

           return
       end subroutine my_func

end module physics
```

solvers.f90 – where the solvers (ODE algorithms) are defined

```
module Solver

    implicit none
    contains

    subroutine euler(n, yin, ynext, t, h, func)
        implicit none
        integer, intent(in) :: n      ! number of ODEs
        real, intent(in)    :: t, h
        real, dimension(n), intent(in)  :: yin
        real, dimension(n), intent(out)  :: ynext
        external      :: func
        integer             :: i
        real, dimension(n) :: k1

        ! call func to obtain the values of dydt

        ! compute ynext using the Euler's method

    end subroutine euler

    subroutine rk2(n, yin, ynext, t, h, func)
        implicit none
        integer, intent(in) :: n      ! number of ODEs
        real, intent(in)    :: t, h
        real, dimension(n), intent(in)  :: yin
        real, dimension(n), intent(out)  :: ynext
        external      :: func
        integer             :: i
        real, dimension(n) :: k1, k2
        real,dimension(n)  :: y2

        ! compute k1 = func(t, yin)

        ! compute y2 = yin + h*k1

        ! compute k2 = func(t+h, y2)

        ! compute ynext

    end subroutine rk2
```

Exercise #1 – implement the Euler's method for simulating the angry bird trajectories

- Take a look at the files and make sure you understand how they work
- Fill in subroutine `my_func` in `physics.f90`
- Implement the Euler's method in subroutine `euler` in `solvers.f90`
- Modify the main program to ***print the results to a file*** using formatted output
- Compile and run the codes by

  ```
  make
  ```

  ```
  ./angry
  ```

- Use your favorite plotting program to plot the trajectory of the angry bird. Overplot the analytical solution and compare

Exercise #1b – implement the RK2 method for simulating the angry bird trajectories

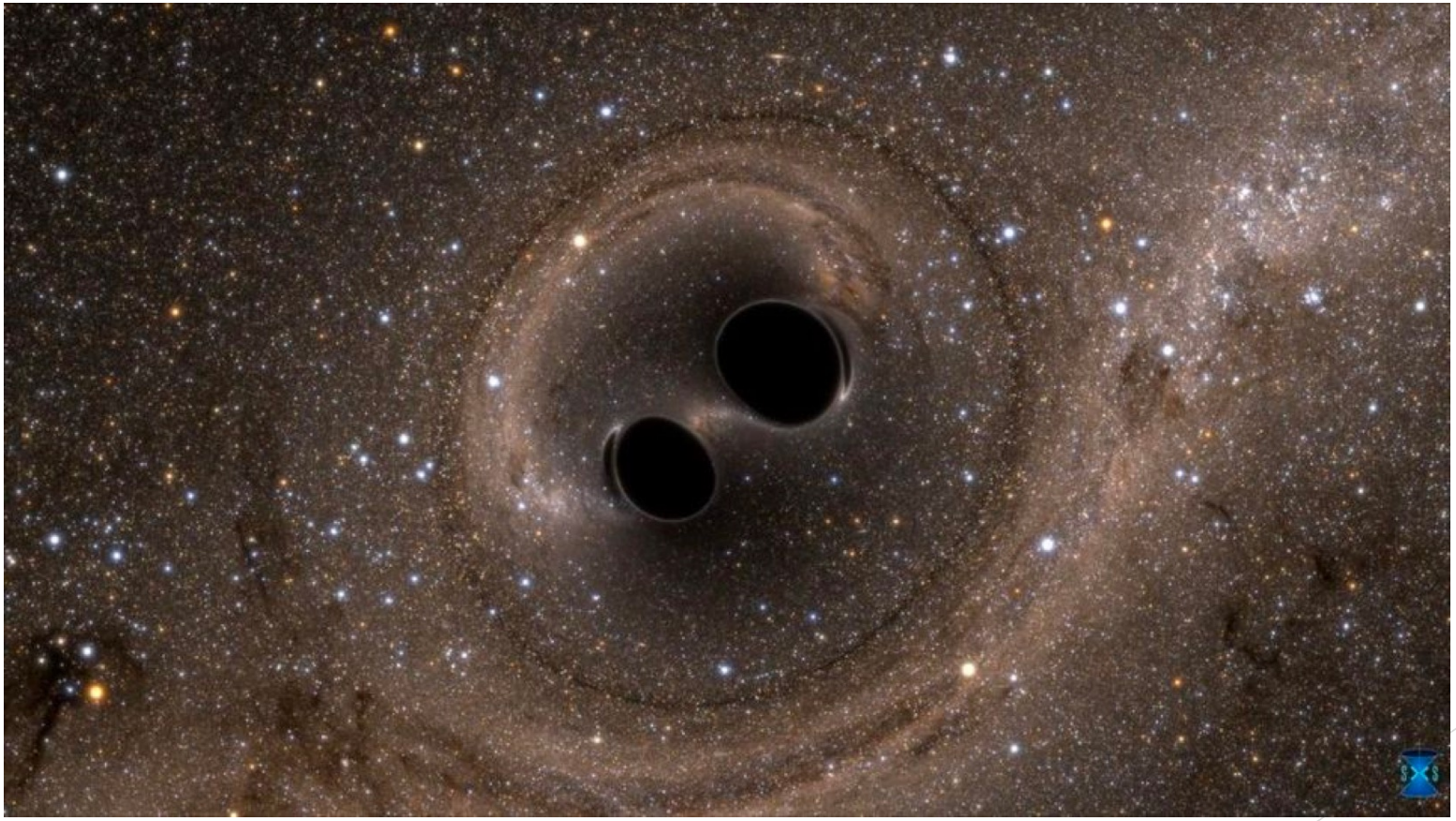
- Modify your code to implement the 2nd-order Runge-Kutta method (RK2) instead
- Fill in subroutine `rk2` in `solvers.f90` (algorithm shown on the right)
- Compile and run the codes by

  ```
  make
  ./angry
  ```

- Plot the trajectory of the angry bird and compare with the analytical solution
- To get the bonus credit, please submit your code and the plots for both the Euler's & RK2 methods to the TAs by end of today (10/27/2022)

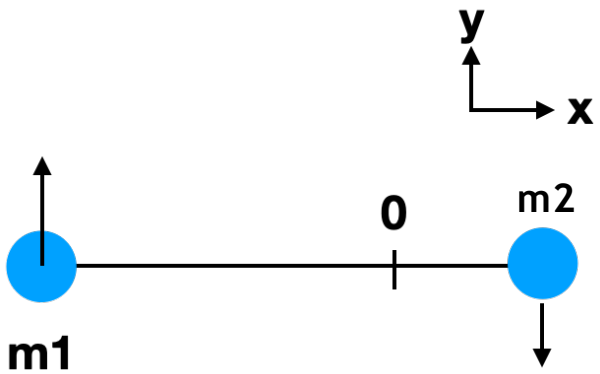
$$k_1 = f(t_k, y_k)$$
$$k_2 = f(t_k + h_k, y_k + h_k k_1)$$
$$y_{k+1} = y_k + \frac{h_k}{2}(k_1 + k_2)$$

Exercise #2 – orbits of binary systems



Exercise #2 – orbits of binary systems

- Let's write a program to simulate the orbits of binary systems
- Assume 2D Newtonian gravity
- Assume zero initial eccentricity (i.e., circular orbits)
- Use Cartesian coordinates with C.M. at origin
- Initial conditions: M_1 = 1 M_{sun}, M_2 = 2 M_{sun}, a = 3AU, $y_1 = y_2 = 0$



$$P^2 = \frac{4\pi^2}{G(M_1 + M_2)} a^3$$

Exercise #2 – implement the Euler's method for simulating binary orbits

- Go to the binary directory and see what files are there:

```
cd ~/astr660/exercise/L7exercise/binary
ls
```

- Take a look at the files and make sure you understand how they work
  - binary.f90 - main program
  - physics.f90 - where the initial conditions are set up and subroutine update is implemented
  - constants.f90 - where the constants are defined
  - Simulation_data.f90 - where the type **"Star"** and other variables are defined
  - output.f90 - where the subroutine record is defined
  - Makefile

Simulation_data.f90 – where the type "Star" and other variables are defined

declare a user-defined "type" which includes 8 attributes for each star

declare an array "stars" to store variables for two stars

```
module Simulation_data

    implicit none

    integer, parameter :: nstars=2 ! number of stars

    type Star
        integer :: id
        real    :: mass
        real    :: x  ! x position
        real    :: y  ! y position
        real    :: vx ! x velocity
        real    :: vy ! y velocity
        real    :: ax ! x acce
        real    :: ay
    end type Star

    type(Star), dimension(nstars) :: stars
    real :: separation
    real :: period
    real :: time

end module Simulation_data
```

Example -- to assign vx for star 1:

stars(1)%vx = ...

Exercise #2 – implement the Euler's method for simulating binary orbits

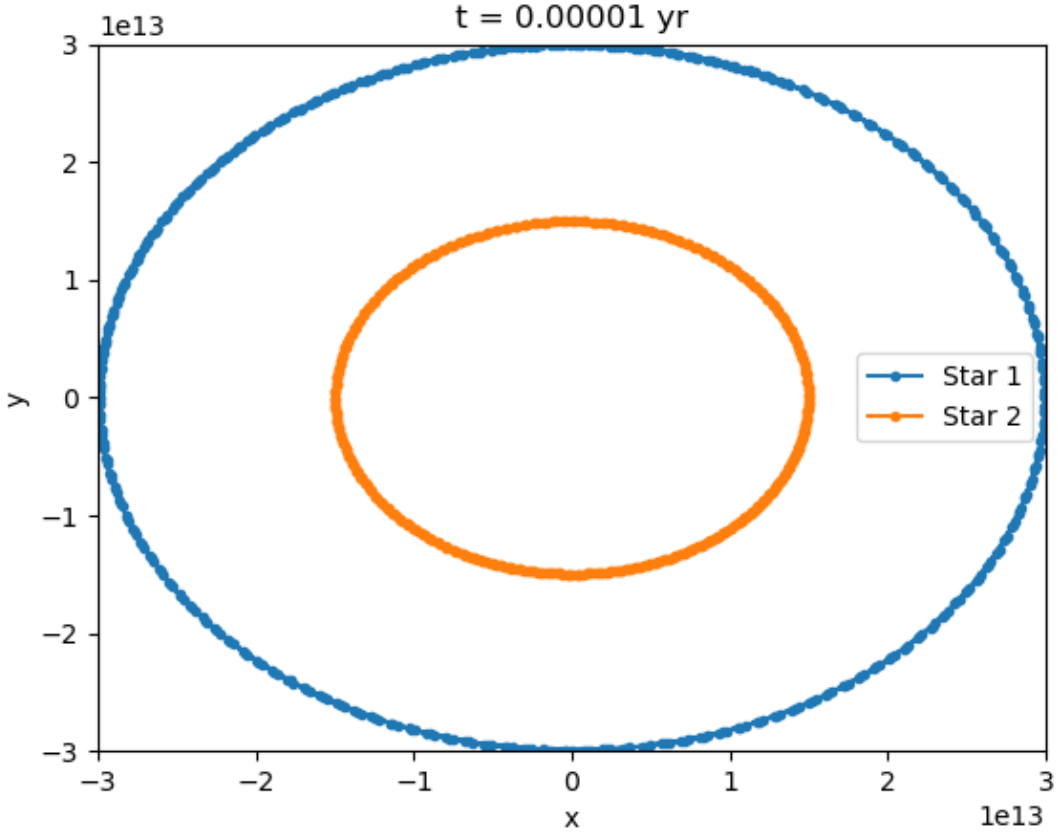
- Set up the initial conditions in subroutine initial in physics.f90
- Update the positions, velocities and ***accelerations*** of the stars using Euler's method in subroutine update in physics.f90
- Compile and run the codes by

  ```
  make
  ```

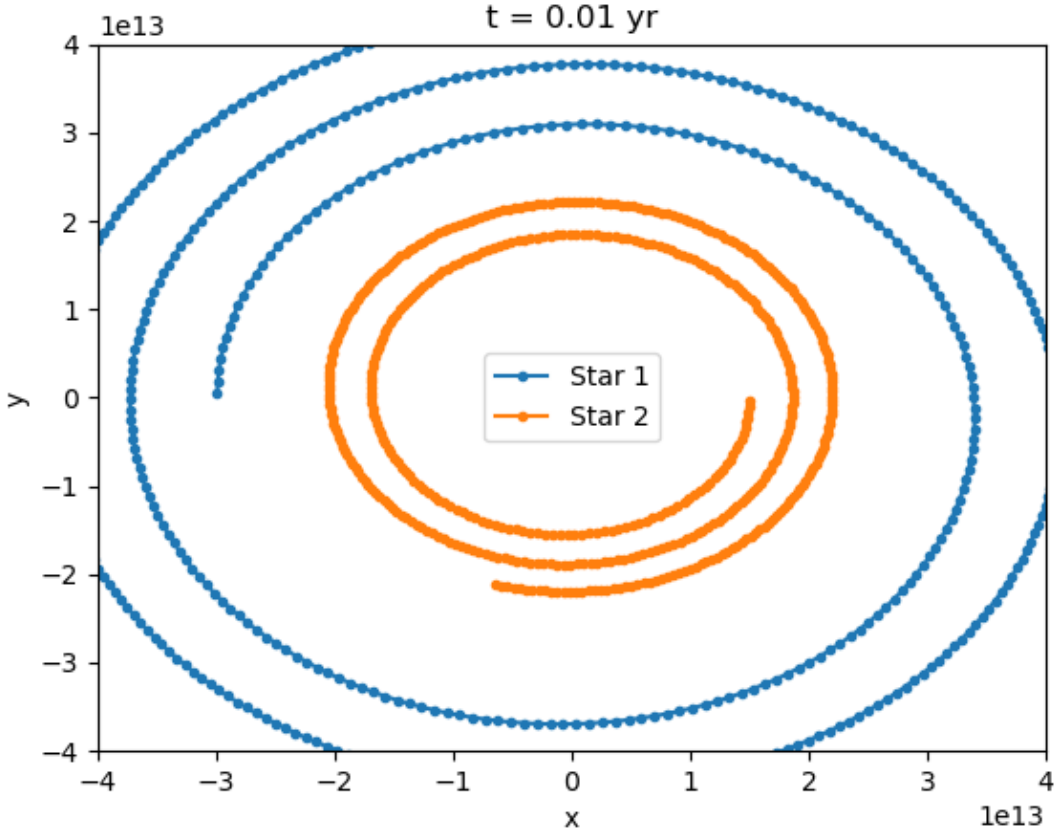
  ```
  ./binary
  ```

- Use your favorite plotting program to plot the trajectory of the two stars
- Try using different timesteps, t = 0.001 and 0.01 yr, and see how the results change

Exercise #2 – implement the Euler's method for simulating binary orbits



Exercise #2 – implement the Euler's method for simulating binary orbits



Typical integrators do not explicitly conserve energy and angular momentum!!

Symplectic integrators

- ***Symplectic (偶對的) integrators*** are designed for solving Hamiltonian systems:

$$\dot{p} = -\frac{\partial H}{\partial q} \quad \text{and} \quad \dot{q} = \frac{\partial H}{\partial p},$$

- The set of position and momentum coordinates *(q, p)* are called ***canonical coordinates***
- Effectively, they conserve the area *dpdq* in phase space during integration
- Could ***conserve total energy*** to much higher precision than typical ODE integrators => widely used in dynamical systems
- For details, see https://en.wikipedia.org/wiki/Symplectic_integrator

Ordinary differential equations -- summary

$$y^{(k)}(t) = f(t, y, y', \ldots, y^{(k-1)})$$

- ***Ordinary differential equations (ODEs)*** are differential equations that involve only one independent variable (e.g., time) and can be solved by broken down to a system of 1st-order ODEs
- Errors occur mainly due to ***truncation errors***. When solving ODEs, one has to care about
  - ***accuracy*** – choosing a numerical method with desired accuracy to reduce local errors per step
  - ***stability*** – choosing a step size that satisfies the stability criterion for the numerical method
  - ***cost*** – choosing a step size as large as allowed to save computational time
- ***Stiff ODEs*** – systems in which the step size is severely limited by the stability criterion

Ordinary differential equations -- summary

$$y^{(k)}(t) = f(t, y, y', \ldots, y^{(k-1)})$$

- Algorithms for solving ODEs: Euler's (x), backward Euler's, Taylor series, Runge-Kutta methods (RK2, RK4), leapfrog...
- ***Order of accuracy for an ODE algorithm:*** to what order of *h* does the approximation agrees with the exact function
- ***Explicit*** methods: use info only at t_k, easier to compute, smaller stability region (requiring smaller step sizes), inefficient for stiff ODEs
- ***Implicit*** methods: use info also at t_{k+1}, require solving nonlinear equations to obtain next solution, larger stability region (sometimes unconditionally stable), suitable for solving stiff ODEs
- ***Symplectic integrators***: algorithm for solving Hamiltonian systems, nearly energy conserving, useful for simulating celestial movements

References & acknowledgements

- Course materials of Computational Astrophysics from Prof. Kuo-Chuan Pan (NTHU)
- Course materials of Computational Astrophysics from Prof. Hsi-Yu Schive (NTU)
- Course materials of Computational Astrophysics and Cosmology from Prof. Paul Ricker (UIUC)
- “Computational Physics” by Rubin H. Landau, Manuel Jose Paez and Cristian C. Bordeianu
- “Scientific Computing – An Introductory Survey” by Michael T. Heath