

Goals for This Lecture:

- Understand why Euler's method sometimes gives inaccurate answers
- Develop some more accurate ODE integration methods
- Develop general purpose ODE codes
- Compare these methods on the orbital problem

Euler's Method for Coupled Systems of ODEs

- Let us develop a shorthand notation for the time at the nth step t^n we will denote $y_i(t^n)$ as y_i^n
- Then we can approximate the derivatives as: $\frac{dy_i}{dt} \approx \frac{y_i^{n+1} - y_i^n}{\Delta t}$
- Thus Euler's method for a set of couple ODEs is:

$$\begin{aligned}y_1^{n+1} &= y_1^n + \Delta t f_1(y_1^n, y_2^n, \dots, y_N^n, t^n) \\y_2^{n+1} &= y_2^n + \Delta t f_2(y_1^n, y_2^n, \dots, y_N^n, t^n) \\&\vdots \\y_N^{n+1} &= y_N^n + \Delta t f_N(y_1^n, y_2^n, \dots, y_N^n, t^n)\end{aligned}$$

- **Exercise: Convince yourself that the orbital equations fit the form described in the previous slide and that the numerical method we used fits into the form shown above**

Implementing a problem independent Euler's Method Code

- We can implement a generic Euler's method code that will work on any system of coupled first order ODEs if we undertake the following steps
- Treat the solution $\{y_i, i=1 \dots N\}$ as a 1-D array (a vector) of length N
 - Have the user supply the number of equations $N = \text{neqns}$ to the code via a subroutine named `prob_size(neqns)`
 - Have the user supply the initial conditions including the initial time t , the timestep size, dt , the time to stop integrating, $tstop$, and the initial value of the solution vector, y , to the code via a subroutine named `prob_init_conds(t,dt,tstop,y)`
 - Have the user supply the right-hand-side for each of the equations via a subroutine that takes in t & y as inputs and returns a vector rhs containing the right-hand-sides. We will call this subroutine `prob_rhs(t,y,rhs)`

Implementing a problem independent Euler's Method Code

```
! Purpose:   Integrate a set of coupled ODEs with Euler's method
! Note: User must supply subroutines prob_size, prob_init_conds, and prob_rhs
!
program ode_euler
implicit none
integer :: neqns           ! Number of equations
integer :: i              ! Implicit DO-loop index
real(kind=kind(1.0d0)), allocatable :: y(:), rhs(:) ! y & r.h.s.
real(kind=kind(1.0d0)) :: t, tstop, dt      ! Time, Stopping time, & timestep
call prob_size(neqns)           ! Get # of equations
allocate(y(neqns))             ! Allocate y & ynew arrays to correct size
allocate(rhs(neqns))
call prob_init_conds(t,tstop,dt,y) ! Get initial conditions
open(unit=17,file='results.dat') ! Open a file for output
do while(t < tstop)           ! Integrate forward in time
  call prob_rhs(t,y,rhs)      ! Evaluate right-hand-side of ODEs
  y = y+dt*rhs                ! Apply Euler's method
  t = t+dt                     ! Increment time
  write(17,'(t2,20(1x,es12.5))') t,(y(i),i=1,neqns) ! Write out results
enddo
close(unit=17)
stop
end program ode_euler
```

The prob_size subroutine

```
! Purpose: Return the number of equations for  
! the orbital problem (through the argument  
! neqns)
```

```
subroutine prob_size(neqns)
```

```
! Number of equations
```

```
implicit none integer, intent(out) :: neqns
```

```
neqns = 4 ! Initialize this to 4
```

```
return
```

```
end subroutine prob_size
```

The prob_init_conds subroutine

**! Purpose: Return the initial conditions for
! the orbital problem**

```
subroutine prob_init_conds(t,tstop,dt,y)
implicit none
real(kind=kind(1.0d0)), intent(out) :: t, tstop, dt, y(4)
t = 0.0d0 ! Initialize t to zero
tstop = 2.0d0 ! Initialize tstop to two years
write(*,'(t2,a20,$)') "Enter timestep size:"
read(*,*) dt
y(1) = 1.0d0 ! Initialize x coordinate
y(2) = 0.0d0 ! Initialize y coordinate
y(3) = 0.0d0 ! Initialize x velocity
y(4) = 6.29d0 ! Initialize y velocity
return
end subroutine prob_init_conds
```

The prob_rhs subroutine

```
! Purpose: Return the right-hand-side for
! the orbital problem
subroutine prob_rhs(t,y0,rhs)
implicit none real(kind=kind(1.0d0)), intent(in) :: t, y0(4)

real(kind=kind(1.0d0)), intent(out) :: rhs(4)
real(kind=kind(1.0d0)) :: x, y, vx, vy
           ! Constants in units of solar masses and AU
real(kind=kind(1.0d0)), parameter :: grav=39.47, msun=1.0

x = y0(1) ! Unpack y0 vector
y = y0(2)
vx = y0(3)
vy = y0(4)
rhs(1) = vx ! r.h.s. of x position equation
rhs(2) = vy ! r.h.s. of y position equation
rhs(3) = -grav*msun*x/(x**2 + y**2)**1.5d0 ! r.h.s. of X vel. equation
rhs(4) = -grav*msun*y/(x**2 + y**2)**1.5d0 ! r.h.s. of X vel. equation
return
end subroutine prob_rhs
```

More Accurate Methods

- The problem with Euler's method is that the right-hand-side of the equations is evaluated at the beginning of the timestep.
- The right-hand-side usually changes over the course of each timestep and we may be getting an inaccurate answer as a result.
- It would be better if we could evaluate the right-hand-side in the middle of the timestep. However, we can't do that unless we know the solution in advance
- Strategy: Use Euler's method to estimate the solution at the midpoint of the timestep. And then use this estimate to evaluate the right-hand-side
- This is called a second order Runge-Kutta method

Coupled Systems of ODEs in Vector Notation

- In order to simplify the description of the second order Runge-Kutta algorithm we use the following vector notation to simplify the equations

$$\mathbf{y} \equiv (y_1, y_2, \dots, y_N)$$

$$\mathbf{f} \equiv (f_1, f_2, \dots, f_N)$$

- Using this notation the original set of ODEs is:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y}, t)$$

- In this notation Euler's method is:

$$\mathbf{y}^{n+1} = \mathbf{y}^n + \Delta t \mathbf{f}(\mathbf{y}^n, t)$$

The Second-order Runge-Kutta Method

- In vector notation we can easily describe the second-order Runge-Kutta algorithm as follows:
- Step 1: $\mathbf{k}_1 = \Delta t \mathbf{f}(\mathbf{y}^n, t)$
- Step 2: $\mathbf{y}^{n+1} = \mathbf{y}^n + \Delta t \mathbf{f}\left(\mathbf{y}^n + \frac{1}{2}\mathbf{k}_1, t + \frac{1}{2}\Delta t\right)$
- Code for this algorithm is an easy modification from our Euler's method code

The Second Order Runge-Kutta Code

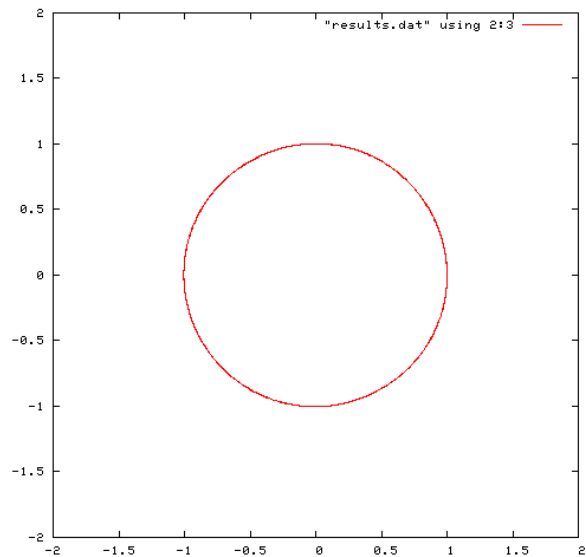
```
program ode_rk2
implicit none
integer :: neqns ! Number of equations
integer :: i ! Implicit DO-loop index
real(kind=kind(1.0d0)), allocatable :: y(:), k1(:), rhs(:) ! y, y1, & r.h.s
real(kind=kind(1.0d0)) :: t, tstop, dt ! Time, Stopping time, & timestep
call prob_size(neqns) ! Get # of equations
allocate(y(neqns)) ! Allocate y & ynew arrays to correct size
allocate(k1(neqns))
allocate(rhs(neqns))
call prob_init_conds(t,tstop,dt,y) ! Get initial conditions
open(unit=17,file='results.dat') ! Open a file for output

do while(t < tstop) ! Integrate forward in time
  call prob_rhs(t,y,rhs) ! Evaluate right-hand-side of ODEs
  k1 = dt*rhs ! Apply first step
  call prob_rhs(t+0.5d0*dt,y+0.5d0*k1,rhs) ! Evaluate right-hand-side at midpoint
  y = y+dt*rhs ! Apply first step
  t = t+dt ! Increment time
  write(17,'(t2,20(1x,es12.5))') t,(y(i),i=1,neqns) ! Write out results
enddo
close(unit=17)
stop
end program ode_rk2
```

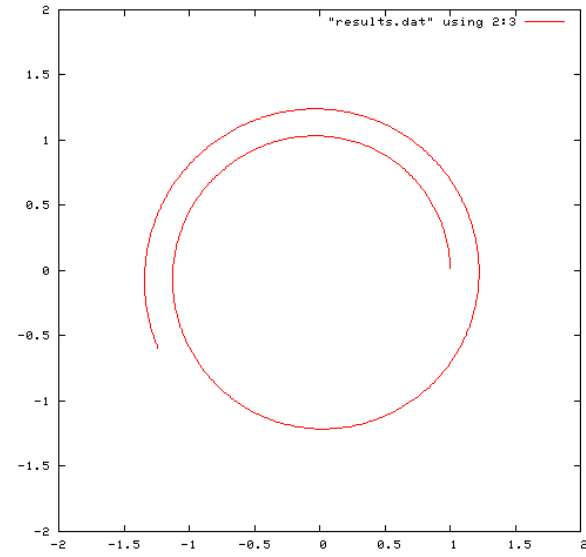
Second-order Runge-Kutta vs. Euler's Method

- Let's now compare the second-order Runge-Kutta and Euler's Method on the orbital problem
- Use a time step of $\Delta t = 0.003$ years

- Second-order RK



- Euler's Method



- Clearly the 2nd-order Runge-Kutta does a lot better

The Fourth-order Runge-Kutta Method

- In practice, the workhorse algorithm for first-order sets of ODEs is the fourth-order Runge-Kutta algorithm which we state here without derivation
- Step 1: $k_1 = \Delta t f(y^n, t)$
- Step 2: $k_2 = \Delta t f(y^n + \frac{1}{2}k_1, t + \frac{1}{2}\Delta t)$
- Step 3: $k_3 = \Delta t f(y^n + \frac{1}{2}k_2, t + \frac{1}{2}\Delta t)$
- Step 4: $k_4 = \Delta t f(y^n + k_3, t + \Delta t)$
- Step 5: $y^{n+1} = y^n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$
- Code for this algorithm is an easy modification from our 2nd-order Runge-Kutta code

The Fourth Order Runge-Kutta Code

```
program ode_rk4
implicit none integer :: neqns           ! Number of equations
integer :: i                             ! Implicit DO-loop index
real(kind=kind(1.0d0)), allocatable :: y(:), rhs(:) ! y, & r.h.s
real(kind=kind(1.0d0)), allocatable :: k1(:), k2(:), k3(:), k4(:)
real(kind=kind(1.0d0)) :: t, tstop, dt   ! Time, Stopping time, & timestep
call prob_size(neqns)                   ! Get # of equations
allocate(y(neqns))                      ! Allocate y & k arrays to correct size
allocate(k1(neqns), k2(neqns), k3(neqns), k4(neqns), rhs(neqns))

call prob_init_conds(t, tstop, dt, y)    ! Get initial conditions
open(unit=17, file='results.dat')       ! Open a file for output

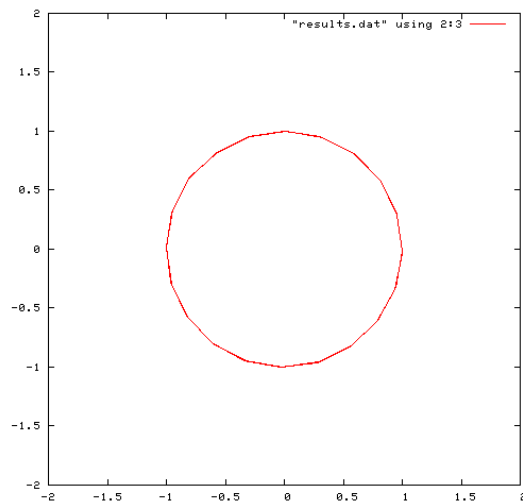
do while(t < tstop)                     ! Integrate forward in time
  call prob_rhs(t, y, rhs)               ! Evaluate right-hand-side of ODEs for k1
  k1 = dt*rhs
  call prob_rhs(t+0.5d0*dt, y+0.5d0*k1, rhs) ! Evaluate right-hand-side of ODEs for k2
  k2 = dt*rhs
  call prob_rhs(t+0.5d0*dt, y+0.5d0*k2, rhs) ! Evaluate right-hand-side of ODEs for k3
  k3 = dt*rhs
  call prob_rhs(t+dt, y+k3, rhs)         ! Evaluate right-hand-side of ODEs for k4
  k4 = dt*rhs
  y = y+(k1+2.0d0*k2+2.0d0*k3+k4)/6.0d0 ! Apply rk4 method
  t = t+dt                               ! Increment time
  write(17, '(t2,20(1x,es12.5))') t, (y(i), i=1, neqns) ! Write out results
enddo

close(unit=17)
stop
end program ode_rk4
```

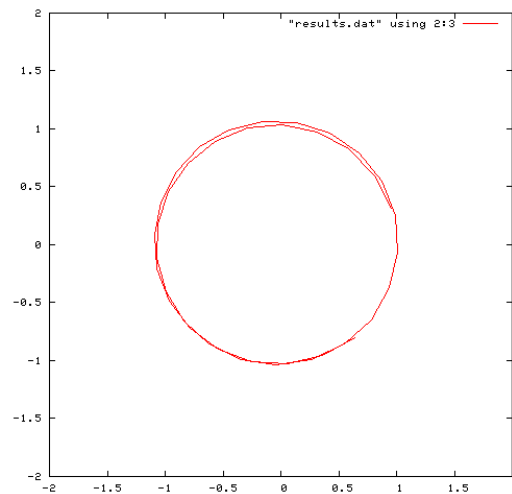
A Comparison of 4th-order RK, 2nd-order RK, and Euler's Method

- Let's now compare the two R-K Methods and Euler's Method on the orbital problem
- Use a time step of $\Delta t = 0.05$ years

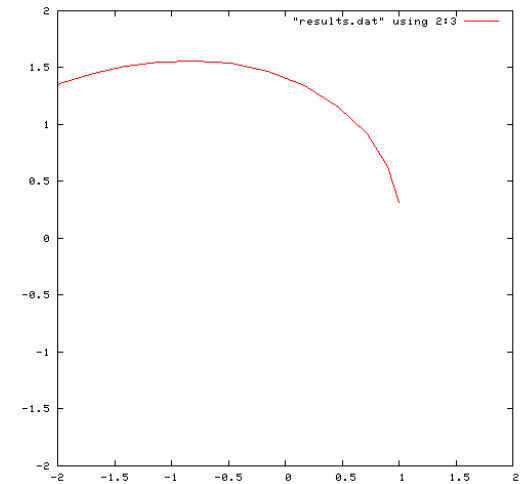
- Fourth-order RK



- Second-order RK



- Euler's Method



- The superiority of the 4th-order Runge-Kutta is apparent
- We'll code this in C++ soon!

Assignment

- Read sections 3.1-3.2 of Savitch