Matlab for Oceanography

Meeting 1

Outline:

Runge-Kutta methods for ODEs:

- Euler
- Heun
- RK4

Lorenz equations

Lotka-Volterra model

Solving Ordinary Differential Equations (ODEs): Runge-Kutta methods

- We will consider differential equations of the general form: $\frac{dy}{dt} = f(t, y)$

- The methods we will discuss have the general form:

  new value = old value + slope*step size

  or $y_{i+1} = y_i + \phi h$

  where ϕ is called the *increment function*, and is used to extrapolate from an old value y_i to a new value y_{i+1}.

Euler's Method

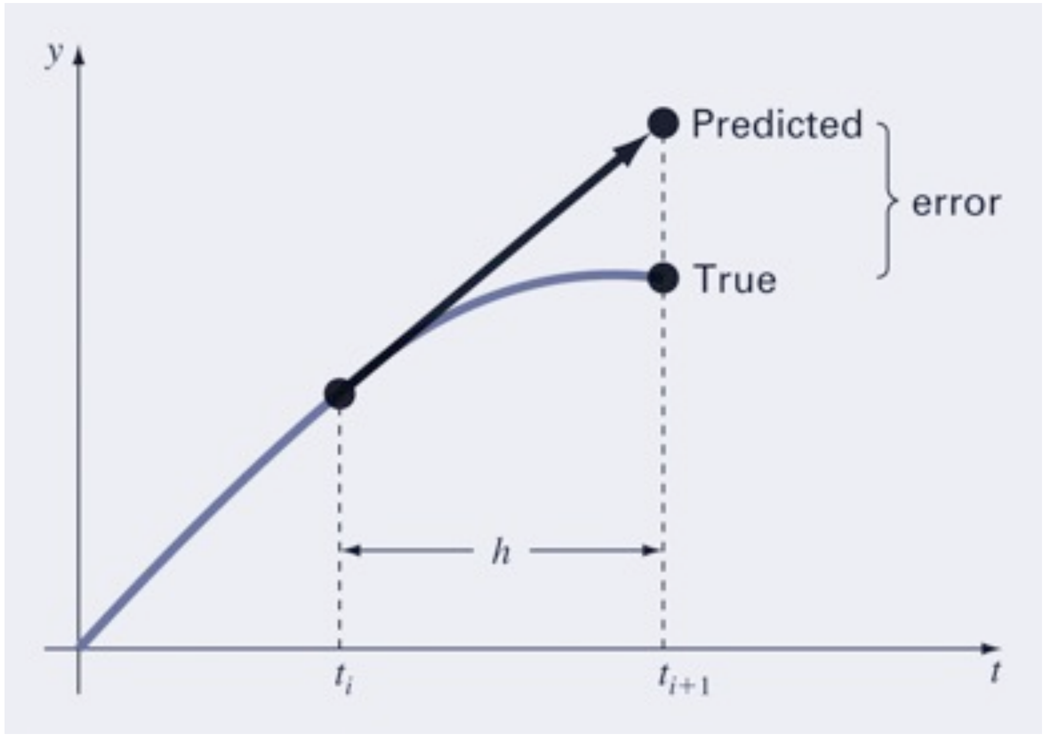
The simplest way of estimating the increment is to take the derivative at t_i

$$\left.\frac{dy}{dt}\right|_{t_i} = f(t_i, y_i)$$

, which estimates the slope of *f* at t_i, and use it as the increment function:

$$\phi = f(t_i, y_i)$$

$$y_{i+1} = y_i + f(t_i, y_i)h$$



Error Analysis for Euler’s Method

The numerical solution of ODEs involves two types of errors:

- *Truncation errors*, caused by the nature of the techniques employed
- *Roundoff errors*, caused by the limited number of significant digits that can be retained

The total, or *global* truncation error can be further split into:

- *local truncation error* that results from the applied method over a single step, and
- *propagated truncation error* that results from the approximations produced during previous steps.

Error Analysis for Euler’s Method

Let’s explore the error using an example:

$$\frac{dy}{dt} = 4\exp(0.8t) - 0.5y$$

Integrate from t = 0 to 4 with a step size of 1 and initial condition y(0) = 2.

We’ll use the function “eulode.m” and compare against the analytical solution

$$y = \frac{4}{1.3}(\exp(0.8t) - \exp(-0.5t)) + 2\exp(-0.5t)$$

Matlab elements to use with “eulode.m”:

Function handle: used to pass the handle to a function as an argument; syntax: `@myfun`

When you have created a function “myfun.m” that evaluates the rhs of our example equation you can pass it to “eulode.m” as follows:

```
>> [t,y] = eulode(@myfun,…)
```

The functions “**nargin**” and “**nargout**” allow you to test with how many input and output arguments a function was called.

“**varargin**” allows you to specify a function with a variable list of input parameters; example:

```
  function y = myfun(varargin)
```

Let’s look at “eulode.m” first.

Then “example.m”, “comparison.m” and “solution.m”.

Note that we could have used *anonymous functions* instead of separate scripts for the “example” and “solution”.

Syntax:

```
>> myfun = @(x) sin(x)+2*cos(x);
>> plot([0:0.1:4],myfun([0:0.1:4]))
```

Look at “comparison_selfcontained.m”

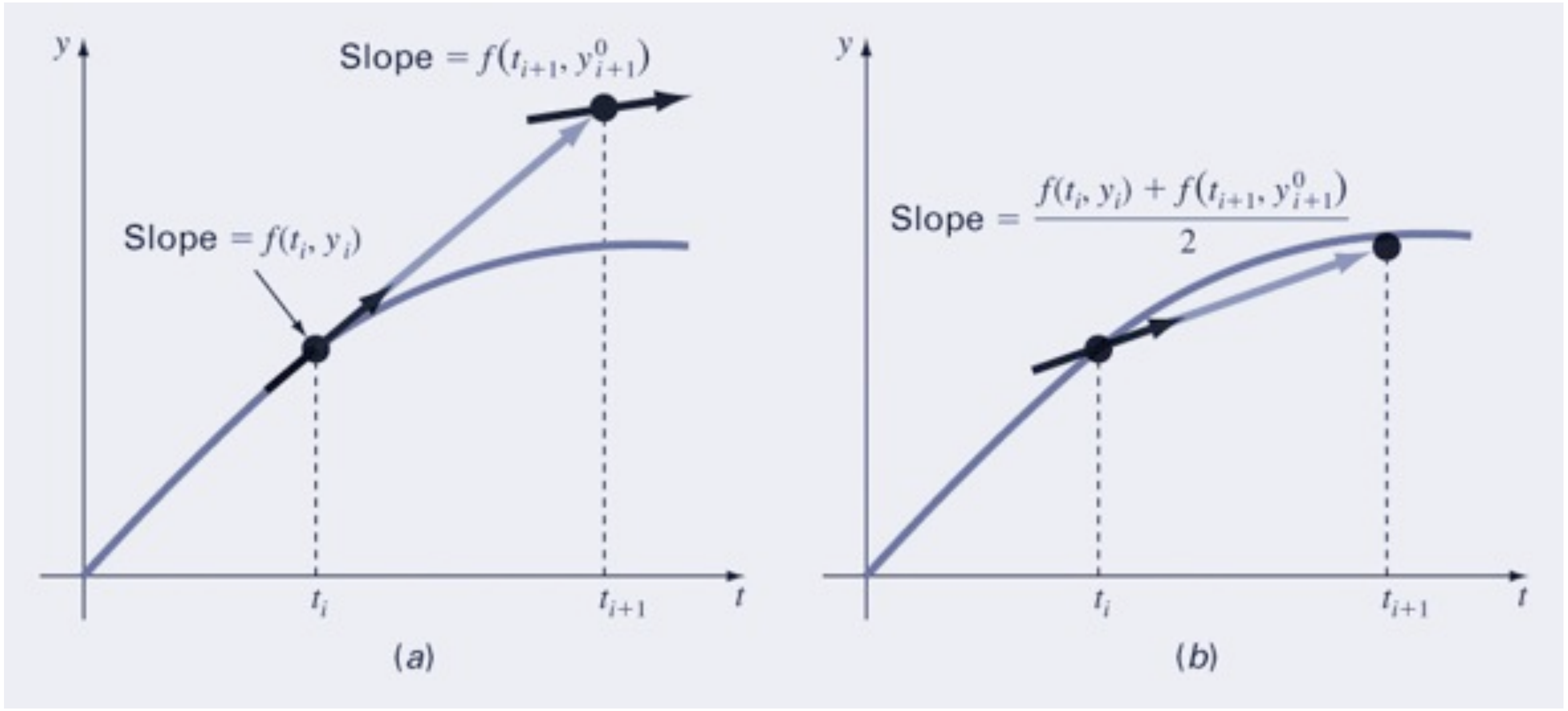
```
 t          y_Euler   y_true   relative error (%)
0            2.0000    2.0000      0
1.0000       5.0000    6.1946    19.2849
2.0000      11.4022   14.8439    23.1863
3.0000      25.5132   33.6772    24.2418
4.0000      56.8493   75.3390    24.5420
```

Output from script “comparison.m”

The error can be reduced in two ways:
- smaller time steps
- modification of the scheme (a fundamental source of error is that the Euler scheme uses the slope at the beginning of the interval)

Heun's Method

One way to improve Euler's method is to determine the derivative at the beginning and predict it for the ending of the interval and average them:



y
Slope $= f(t_{i+1}, y^0_{i+1})$
Slope $= f(t_i, y_i)$
t_i
t_{i+1}
t
(a)
y
Slope $= \frac{f(t_i, y_i) + f(t_{i+1}, y^0_{i+1})}{2}$
t_i
t_{i+1}
t
(b)

In Euler's method we use the slope at the beginning of the interval

$$y_i' = f(t_i, y_i)$$

to extrapolate to y_{i+1}

$$y_{i+1}^0 = y_i + f(t_i, y_i)h$$

(and we would be done).

In Heun's method this is just an intermediate prediction. We now use it to estimate the slope at the end of the interval

$$y_{i+1}' = f(t_{i+1}, y_{i+1}^0)$$

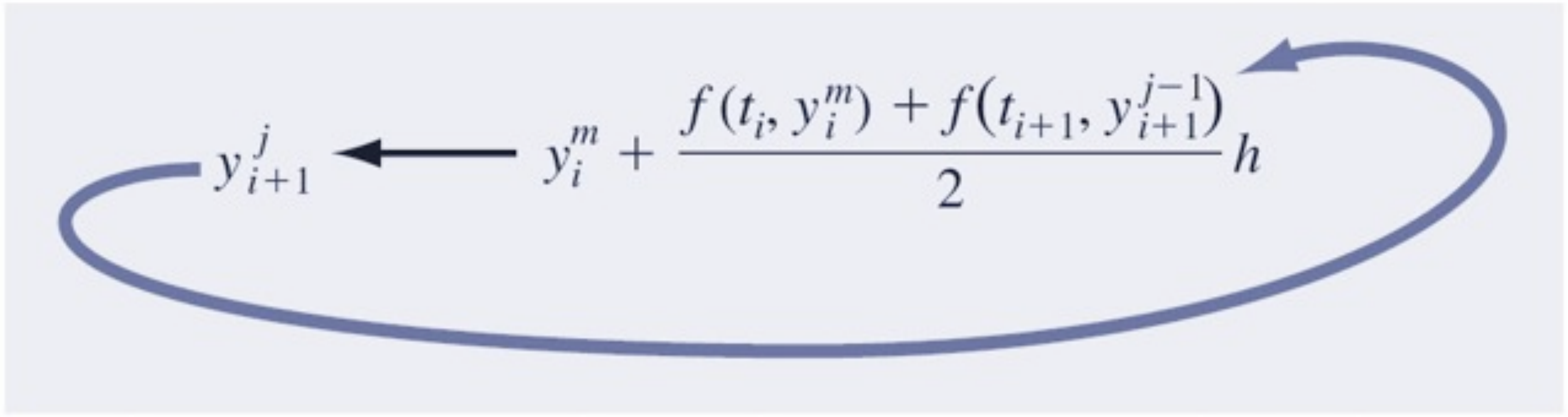
and then calculate the average of both slopes for the increment function:

$$y_{i+1} = y_i + \frac{f(t_i, y_i) + f(t_{i+1}, y_{i+1}^0)}{2}h$$

The 2nd step

$$y_{i+1} = y_i + \frac{f(t_i, y_i) + f(t_{i+1}, y_{i+1}^0)}{2} h$$

can be iterated



$$y_{i+1}^{j} \longleftarrow y_i^m + \frac{f(t_i, y_i^m) + f(t_{i+1}, y_{i+1}^{j-1})}{2} h$$

Both, Euler's and Heun's methods are simple Runge-Kutta methods (1st order and 2nd order).

Runge-Kutta Methods

For RK methods, the increment function ϕ can generally be written as:

$$\phi = a_1 k_1 + a_2 k_2 + \cdots + a_n k_n$$

where the *a*'s are constants and the *k*'s are estimates of the slope at different positions

$$\begin{aligned}
k_1 &= f(t_i, y_i) \\
k_2 &= f(t_i + p_1 h, y_i + q_{11} k_1 h) \\
k_3 &= f(t_i + p_2 h, y_i + q_{21} k_1 h + q_{22} k_2 h) \\
&\vdots \\
k_n &= f(t_i + p_{n-1} h, y_i + q_{n-1,1} k_1 h + q_{n-1,2} k_2 h + \cdots + q_{n-1,n-1} k_{n-1} h)
\end{aligned}$$

where the *p*'s and *q*'s are constants.

The coefficients are typically chosen by setting the general equations equal to the Taylor Series, which yields an underdetermined system for $n>1$.

Some coefficients have to be chosen. Hence, there are infinitely many RK schemes for each $n>1$.

For $n=1$ we get Euler.

For $n=2$ and $a_1 = a_2 = 0.5$ and $p_1 = q_{11} = 1$ we get Heun.

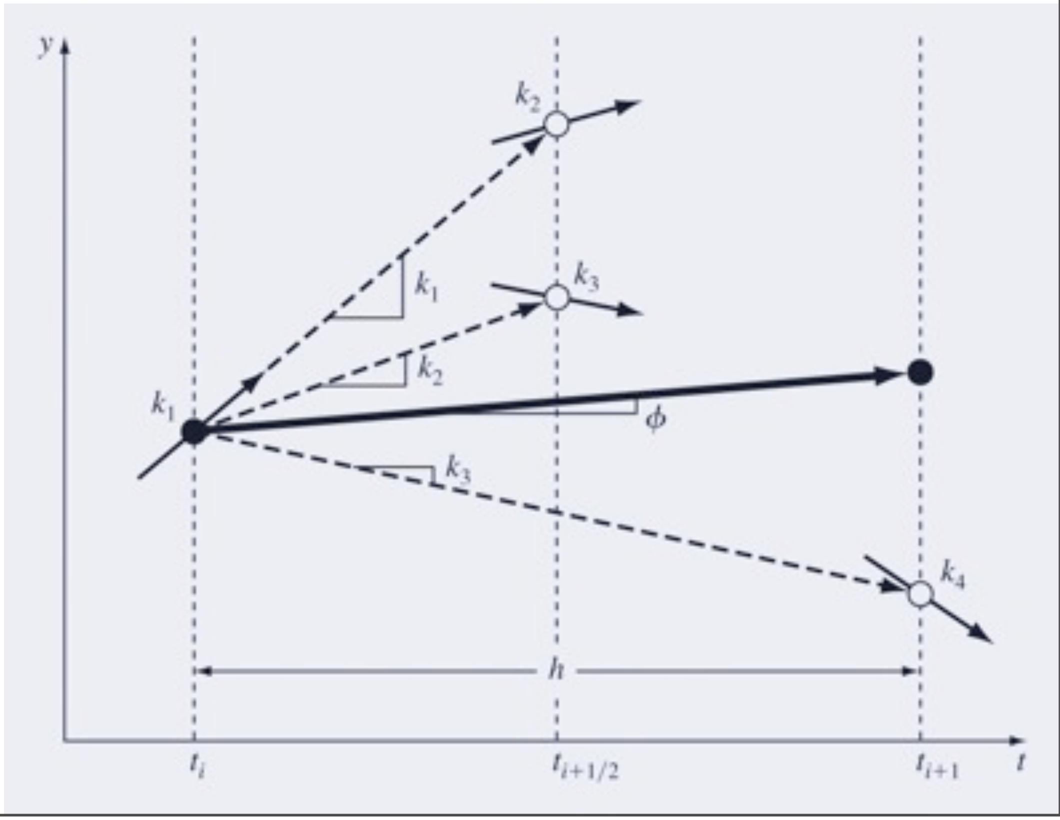
Classical Fourth-Order RK Method

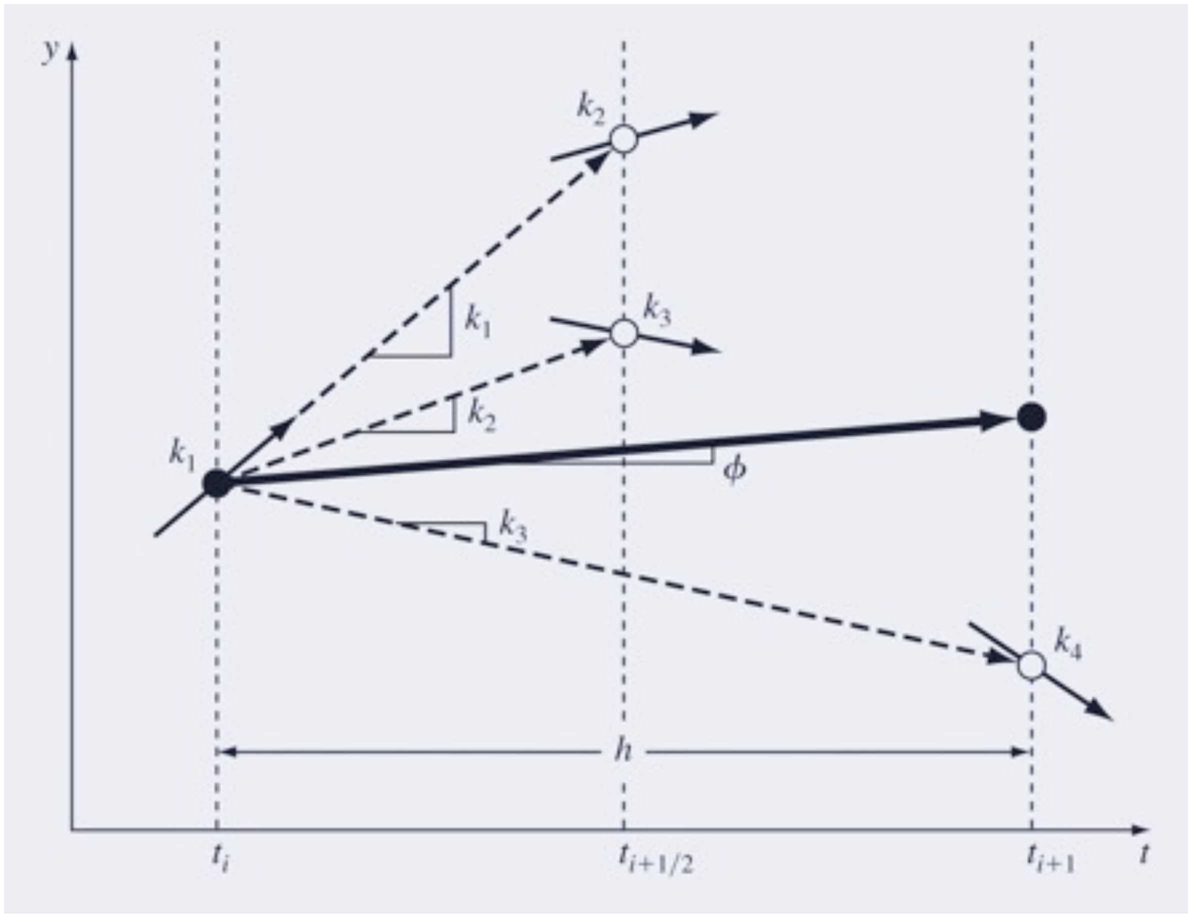
The most popular RK methods are fourth-order, and the most commonly used form is:

$$y_{i+1} = y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)h$$

where:

$$k_1 = f\left(t_i, y_i\right)$$
$$k_2 = f\left(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1 h\right)$$
$$k_3 = f\left(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2 h\right)$$
$$k_4 = f\left(t_i + h, y_i + k_3 h\right)$$





All *k*s are approximations of the slope.
ϕ is a weighted average of these slopes.

Implement the 4th-order RK method by modifying “eulode.m” appropriately and solve our example.

Compare errors with our previous approximation and with the true solution for this method.

t	y_Euler	y_RK4	y_true	Eul.rel.err.(%)	RK4rel.err.(%)
0	2.0000	2.0000	2.0000	0	0
1.0000	5.0000	6.2010	6.1946	19.2849	0.1034
2.0000	11.4022	14.8625	14.8439	23.1863	0.1250
3.0000	25.5132	33.7213	33.6772	24.2418	0.1312
4.0000	56.8493	75.4392	75.3390	24.5420	0.1330

See my scripts “rk4ode.m” and “comparison2.m”

What we’ve done so far was meant to explain “what’s under the hood” of the Matlab solvers for ODEs, e.g.:
- ode23
- ode45

(implementations of RK solvers of different order)

Syntax:

```
[t, y]=ode23(@function,tspan,y0)
```

where
  y is a vector of solutions,
  t is a vector of the independent variable values corresponding to y,
  you must provide rhs as a function (as before),
  tspan is the integration interval, e.g. [ti tf] or [ti:h:tf], and
  y0 is the vector of initial condition.

Let’s repeat our example with ode23 and ode45.

t	y_Euler	y_RK4	y_ODE23	y_ODE45	y_true	Eul.RE(%)	RK4 RE(%)	ODE23 RE(%)	ODE45 RE (%)
0	2.0000	2.0000	2.0000	2.0000	2.0000	0	0	0	0
1	5.0000	6.2010	6.1935	6.1946	6.1946	19.2849	0.1034	0.0186	0.0000
2	11.4022	14.8625	14.8392	14.8439	14.8439	23.1863	0.1250	0.0316	0.0000
3	25.5132	33.7213	33.6652	33.6772	33.6772	24.2418	0.1312	0.0355	0.0000
4	56.8493	75.4392	75.3150	75.3390	75.3390	24.5420	0.1330	0.0318	0.0000

Notice the dramatic improvement in accuracy.

See script “comparison3.m”

All the methods discussed so far apply also to systems of ODEs

$$\frac{dx_1}{dt} = f_1(t, \vec{x})$$
$$\frac{dx_2}{dt} = f_2(t, \vec{x})$$
$$\frac{dx_3}{dt} = f_3(t, \vec{x})$$

We will now consider two examples:

- Lorenz equations
- Lotka-Volterra predator-prey model

The reason why weather prediction is so difficult is that atmospheric models (or the DEs representing them) have properties that make them behave chaotically. The results will change wildly for small changes in initial conditions, numerical schemes, etc.

Ed Lorenz (MIT) described this phenomenon in 1961.

He developed these simple equations to relate the intensity of atmospheric fluid motion *x* to temperature variations *y* and *z* in the horizontal and vertical directions:

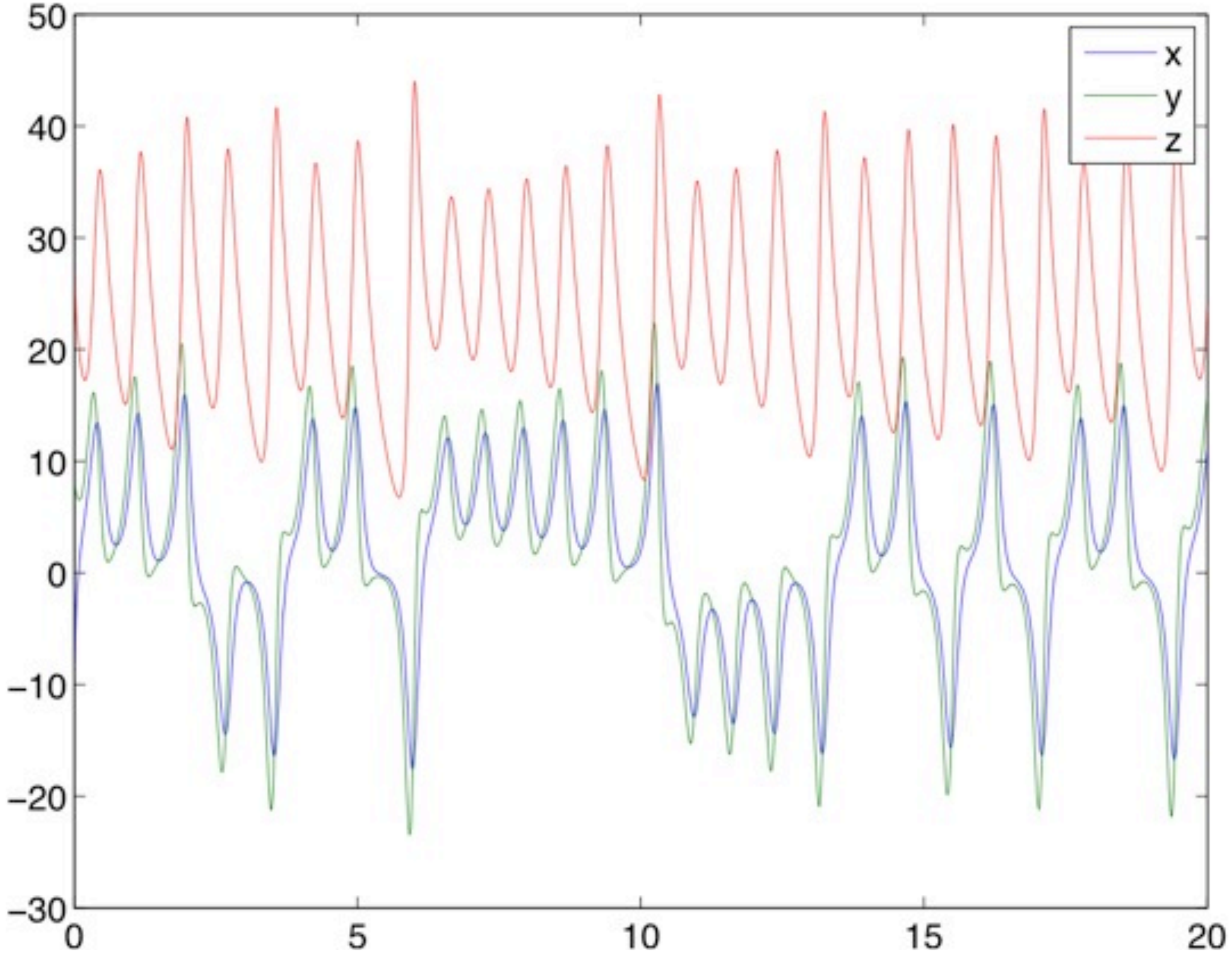
$$\begin{aligned}
\frac{dx}{dt} &= 10(y-x) \\
\frac{dy}{dt} &= -xz + 28x - y \\
\frac{dz}{dt} &= xy - \frac{8}{3}z
\end{aligned}$$

$$\begin{aligned}\frac{dx}{dt} &= 10(y-x)\\ \frac{dy}{dt} &= -xz+28x-y\\ \frac{dz}{dt} &= xy-\frac{8}{3}z\end{aligned}$$

We'll solve this with Matlab's ode45 solver for initial conditions:

x(0) = -8,
y(0) = 8,
z(0) = 27

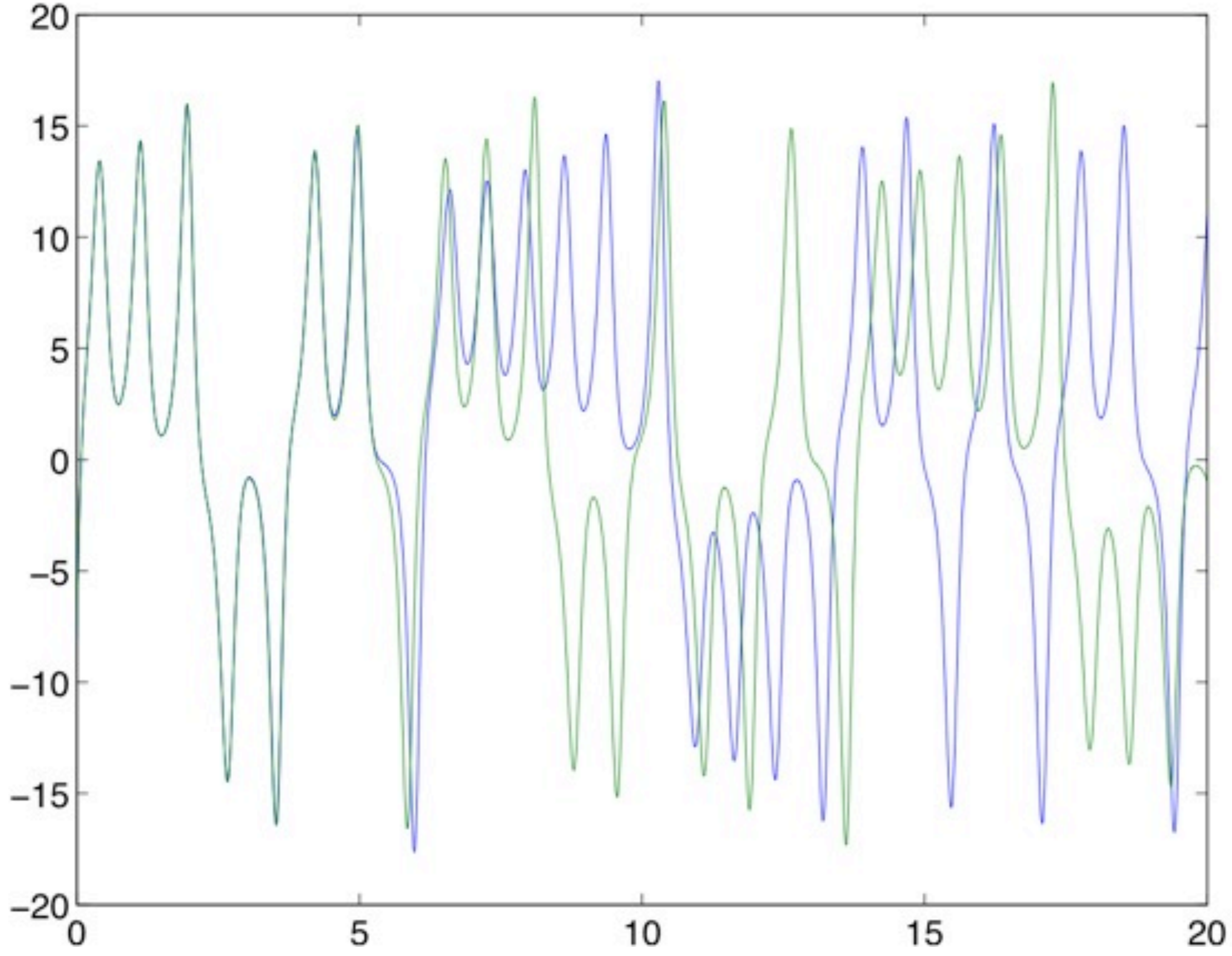
and for t=0 to 20


50
40
30
20
10
0
−10
−20
−30
0
5
10
15
20
x
y
z

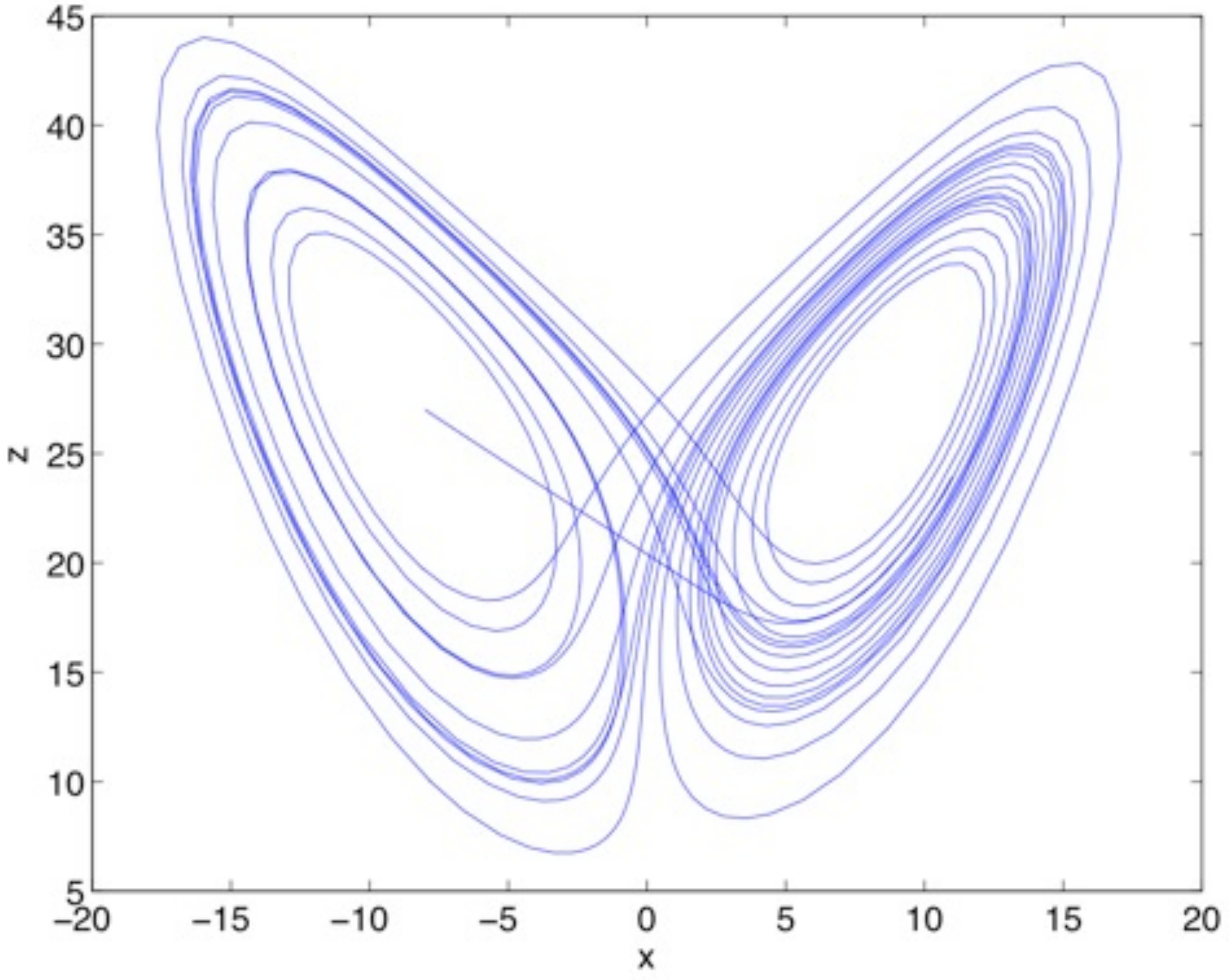
Now rerun with slightly different initial conditions:

$x(0) = -8.01$,
$y(0) = 8$,
$z(0) = 27$

and compare x for both cases.



Note how the solutions diverge after t=5.

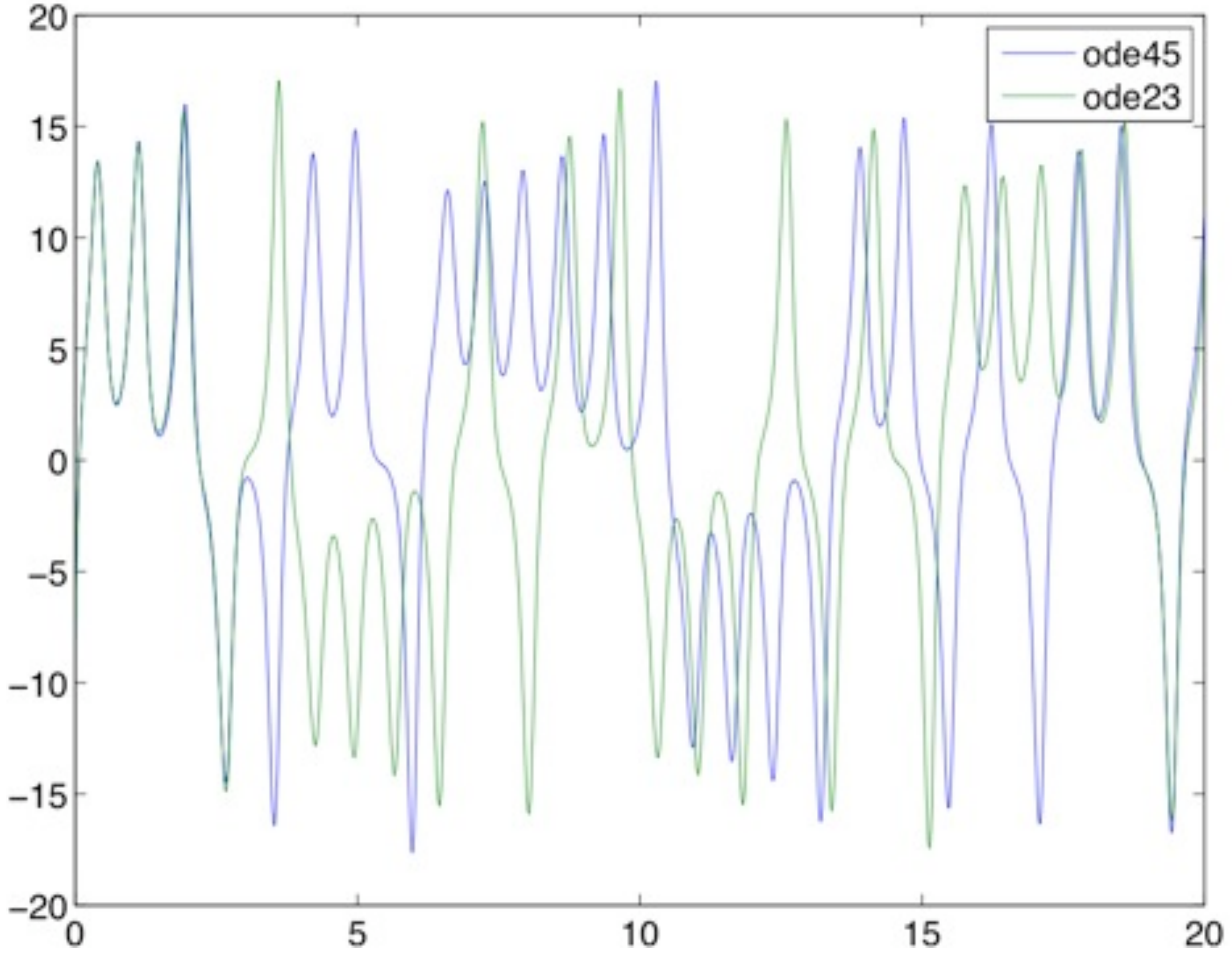


Notice how the solution has two orbits and a critical point (also called strange attractor) where it jumps between them.

Now compare solutions determined with ode45 and ode23 for the same initial conditions:

$x(0) = -8$,
$y(0) = 8$,
$z(0) = 27$

and for $t=0$ to 20



Notice how small differences in the numerical schemes lead to different behavior.

See scripts “lorenz.m” and “solving_lorenz.m”

This behavior led Ed Lorenz to the conclusion that long-range weather forecasts might be impossible.

Note, that I am not making statements about climate forecasts!

Passing parameters to an ODE solver

In the above example we hard-coded the parameters into the function. In real applications we will probably want to change parameters frequently.

We can modify “lorenz.m” as in “lorenz2.m”:

```
function f = lorenz2(t,x,sigma,rho,beta)
```

And call the ode solver as follows:

```
>> [t, x] = ode45(@lorenz2,[0 20],xini,[],
10,28,8/3);
```

Note that the 4th argument is reserved for options (see help ode45). We need to use [] in our case because we don’t want to specify any options.

Lotka-Volterra model

The earliest predator-prey models were developed independently in the early 20th century by Italian mathematician Vito Volterra and American biologist Alfred Lotka.

The simplest version:

$$\begin{aligned} \frac{\partial x}{\partial t} &= ax - bxy \\ \frac{\partial y}{\partial t} &= -cy + dxy \end{aligned}$$

where *x* and *y* represent the biomass of prey and predators, respectively,
a is the prey growth rate, *c* the predator death rate,
b and *d* are predation-related rates describing prey loss and predator growth.

Solve the Lotka-Volterra model with the Euler method and ODE45 passing the parameters externally.

Use the following values:
$a = 1.2$, $b = 0.6$, $c = 0.8$, and $d = 0.3$
Initial conditions of $x = 2$, $y = 1$
Integrate from $t = 0$ to 40 using step size of 0.0625

Plot the results to visualize how the dependent variables evolve over time. In addition, graph the dependent variables against each other.

Is the time step appropriate for the Euler method?

Note: You will have to modify “eulode.m” for a system of ODEs.

See “predprey.m”, “eulsys.m” and “predprey_driver.m”