# Overview

# Recap: The midpoint method

First order ODE

$$\frac{dy}{dx} = f(x, y)$$

Take a 'trial step' to the midpoint:

$$y_{\text{mid}} = y_n + \frac{\varepsilon}{2} f(x_n) \qquad \text{[Looks like Euler!]}$$

Then use this to evaluate $f$ at the midpoint:

$$y_{n+1} = y_n + \varepsilon f(x_{\text{mid}}, y_{\text{mid}})$$

# Runge–Kutta methods

Midpoint method: Take a trial step to evaluate rhs $f(x, y)$ at midpoint
$\longrightarrow$ improved accuracy

There are many ways of evaluating $f(x, y)$ that agree to leading order.
Use this to eliminate higher order error terms

## Most used: Fourth order Runge–Kutta

$$y_{n+1} = y_n + \frac{\varepsilon}{6}(F_1 + 2F_2 + 2F_3 + F_4) \qquad \text{with}$$

$$F_1 = f(x_n, y_n) \qquad \text{'Euler term'}$$

$$F_2 = f(x_n + \frac{\varepsilon}{2}, y_n + \frac{\varepsilon}{2}F_1) \qquad \text{'midpoint term'}$$

$$F_3 = f(x_n + \frac{\varepsilon}{2}, y_n + \frac{\varepsilon}{2}F_2) \qquad \text{'modified midpoint term'}$$

$$F_4 = f(x_n + \varepsilon, y_n + \varepsilon F_3) \qquad \text{'endpoint term'}$$

Step error: $\mathcal{O}(\varepsilon^5)$ Total error: $\mathcal{O}(\varepsilon^4)$

Note: For $y'(x) = f(x)$ RK4 gives Simpson's rule!

# Why Runge–Kutta?

RK4 is preferable over Euler (midpoint) if improved accuracy allows us to take 4 (2) times longer steps to compensate for more function evaluations.
This is usually the case (but not always)
Runge–Kutta does not guarantee stability, but

1. Accuracy is sometimes more important (short runs)
2. Instability can be delayed by higher accuracy
3. Runge–Kutta methods work for all types of ODEs. They can be made into black-box ODE solvers

# Reduction to first order ODEs

We can transform any ODE into a set of coupled first order equations, like we did with the second order equation

## Example ($N$'th order equation)

$$\frac{d^N y}{dt^N} = f(y,t) \quad \Longleftrightarrow \quad \begin{cases} \frac{dy_1}{dt} & = y_2 \quad [\equiv y'(t)] \\ \frac{dy_2}{dt} & = y_3 \quad [\equiv y''(t)] \\ & \vdots \\ \frac{dy_N}{dt} & = f(y_1, t) \end{cases}$$

## Example (Kepler equations)

$$\begin{array}{rcl} \ddot{r} & = & r\dot{\theta}^2 - \frac{GM}{r^2} \\ r^2\ddot{\theta} + 2r\dot{r}\dot{\theta} & = & 0 \end{array} \quad \Longleftrightarrow \quad \begin{cases} \frac{dy_1}{dt} \equiv \frac{dr}{dt} & = y_2 \\ \frac{dy_2}{dt} \equiv \frac{d\dot{r}}{dt} & = y_1 y_4^2 - \frac{GM}{y_1^2} \\ \frac{dy_3}{dt} \equiv \frac{d\theta}{dt} & = y_4 \\ \frac{dy_4}{dt} \equiv \frac{d\dot{\theta}}{dt} & = -2\frac{y_2 y_4}{y_1} \end{cases}$$

# ODEs in vector form

We can write the Euler algorithm (and midpoint and RK4) in vector form:

$$\begin{aligned} \frac{dy_1}{dx} &= f_1(x, y_1, \ldots, y_n) \\ \frac{dy_2}{dx} &= f_2(x, y_1, \ldots, y_n) \\ &\ldots \\ \frac{dy_n}{dx} &= f_n(x, y_1, \ldots, y_n) \end{aligned} \quad \longrightarrow \quad \frac{d\mathbf{y}}{dx} = \mathbf{f}(x, \mathbf{y})$$

Implement $\mathbf{f}(x, \mathbf{y})$ as a vector function and make algorithm call that function

# Pendulum

The equation of motion for a simple pendulum is

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

## Step 1: Dimensionless time variable

$$\tau = \sqrt{g/\ell}\,t \quad \Longrightarrow \quad \frac{d^2\theta}{d\tau^2} = -\sin\theta$$

## Step 2: Reduce to first order equations

$$\begin{array}{rcl} \frac{d\theta}{d\tau} & = & \omega \\ \frac{d\omega}{d\tau} & = & -\sin\theta \end{array} \quad \Longrightarrow \quad \frac{d\mathbf{y}}{d\tau} = \begin{pmatrix} \omega \\ -\sin\theta \end{pmatrix} = \begin{pmatrix} y_2 \\ -\sin y_1 \end{pmatrix}$$

## MatLab implementation

```
function dy = pendulum(t,y)
dy(1) = y(2);
dy(2) = -sin(y(1));
```

# Integrating pendulum equations

## Euler method

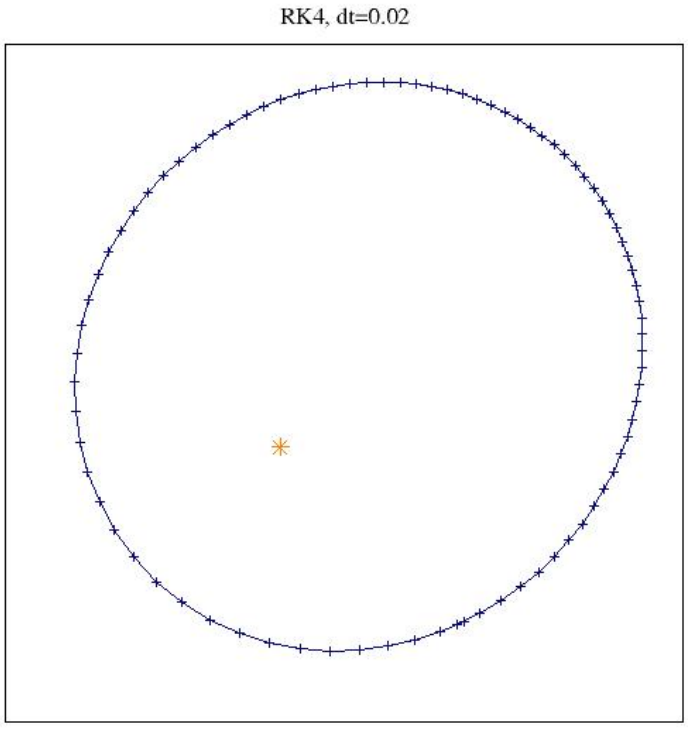
```
y(1,:) = [theta0 omega0];
t = 0:dt:tmax;
N = length(t);
for n=1:N-1
  y(n+1,:) = y(n,:) + dt*pendulum(t(n),y(n,:));
end
```

## Midpoint method

```
for n=1:N-1
  ymid     = y(n,:) + 0.5*dt*pendulum(t(n),y(n,:));
  y(n+1,:) = y(n,:) + dt*pendulum(t(n)+0.5*dt,ymid);
end
```

# Slow vs fast motion

Example: Kepler problem




- Near perihelion: fast motion
- $r(t), \theta(t)$ vary rapidly
- need small time steps
- Near aphelion: slow motion
- can afford bigger time steps

Solution: Let computer decide what stepsize is needed!

# Adaptive stepsize

Let computer decide stepsize depending on what precision we want

- Estimate error at current $\Delta t$
- Increase $\Delta t$ when error estimate 'too small'
- Decrease $\Delta t$ when error estimate 'too big'

## How to estimate the error?

Two approaches:

- Take two steps with $\Delta t/2$, compare with one step with $\Delta t$
- Find difference between higher-order step and lower-order (eg RK4 and midpoint)

# Adaptive stepsize

## What is too small, too big?

- A fixed number is not useful:
  in Kepler problem $\theta$ is in radians but $r$ could be in metres!
- Try to keep

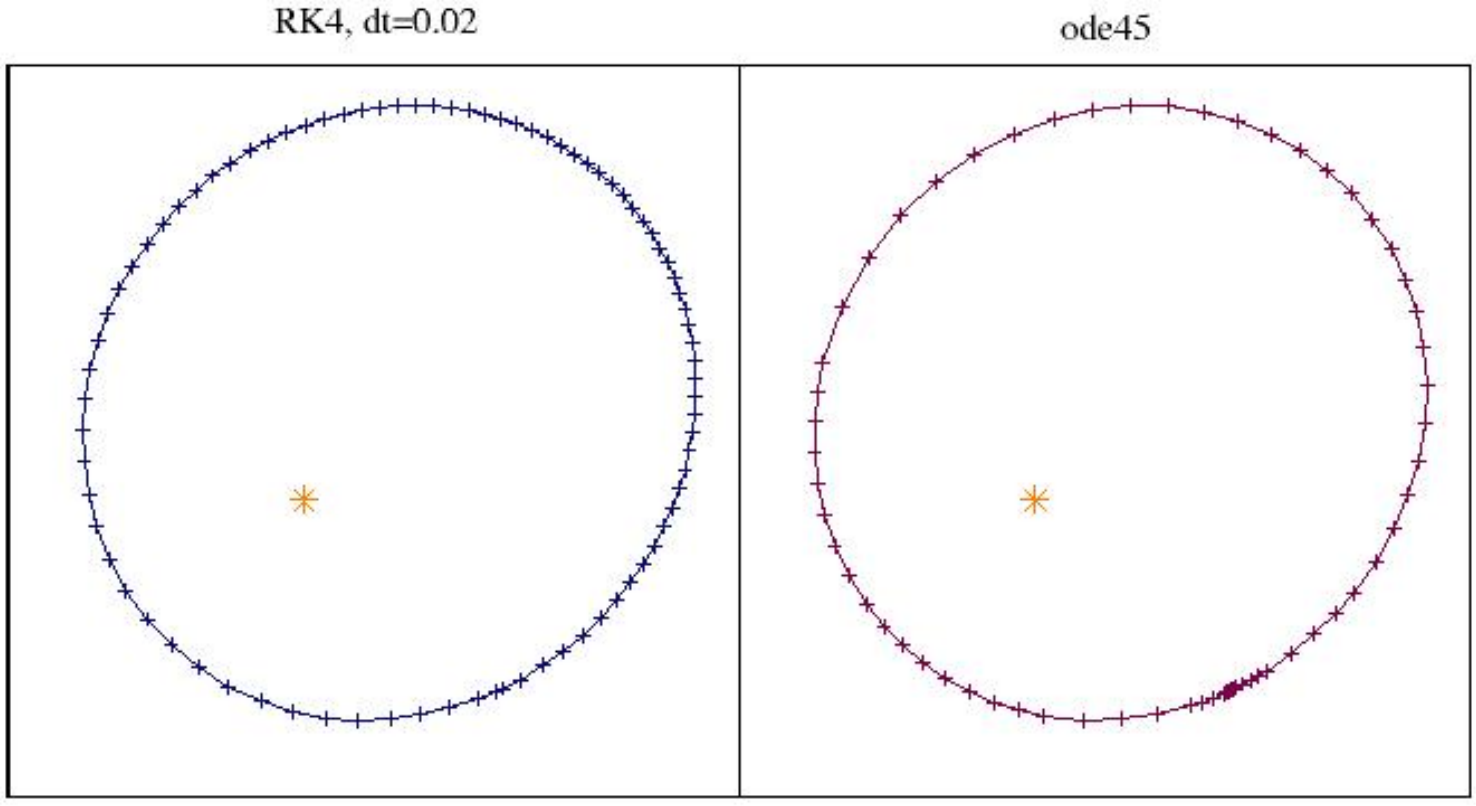
$$\left|\frac{\Delta r}{r}\right| \sim \left|\frac{\Delta \theta}{\theta}\right| \sim \Delta$$

  — relative tolerance
- $\theta$ can become zero, so we also need absolute tolerance $\Delta\theta > \Delta^{\theta}_{\text{min}}$
- Use physical insight!

The 'optimal' step size can be estimated using Taylor expansion:

$$\Delta t_{\text{opt}} = \Delta t \left|\frac{\Delta_{\text{rel}}}{\Delta}\right|^{1/N}, \qquad N = \text{order of algorithm}$$

# Fixed vs adaptive stepsize



# Subjects not covered

- Stiff sets of equations
- Implicit methods
- Richardson extrapolation
- Boundary value problems — next!

# Summary

- Runge–Kutta methods: systematic improvement in accuracy
  - Midpoint method = second order Runge–Kutta
  - Fourth-order Runge–Kutta very widely used
  - Higher order used in adaptive stepsize algorithms
- Adaptive stepsize
  - Allows us to vary the stepsize locally during integration
  - Based on estimating the error in the algorithm
- Vectorising coupled ODEs and using function handles allows us to write universal ODE solvers