# Computational Physics
# Lecture 12 – Solving Initial Value Problems

Pat Scott

Department of Physics, Imperial College

November 9, 2018

Slides available from https://bb.imperial.ac.uk/

# Outline

# Goals

The point of this lecture is to get you to the stage where you can

- Easily identify initial value problems
- Understand the difference between predictor-corrector, leapfrog and Runge-Kutta methods for solving them
- Implement all these methods on a computer
- Choose wisely between implicit and explicit differencing schemes

# Main types of boundary conditions (recap)

1. $\vec{y}(x_s) = y_{\rm start}$
   - Function and all derivatives are defined at some $x_s$
   - All that remains is to evolve the system forwards and/or backwards from $x_s$ to get $\vec{y}(x)$ for all $x$ of interest
   - This is an **Initial Value Problem**

# Main types of boundary conditions (recap)

1. $\vec{y}(x_s) = y_{\text{start}}$
   - Function and all derivatives are defined at some $x_s$
   - All that remains is to evolve the system forwards and/or backwards from $x_s$ to get $\vec{y}(x)$ for all $x$ of interest
   - This is an **Initial Value Problem** – covered today

# Main types of boundary conditions (recap)

1. $\vec{y}(x_\mathrm{s}) = y_\mathrm{start}$
   - Function and all derivatives are defined at some $x_\mathrm{s}$
   - All that remains is to evolve the system forwards and/or backwards from $x_\mathrm{s}$ to get $\vec{y}(x)$ for all $x$ of interest
   - This is an **Initial Value Problem** – covered today
2. Some components of $\vec{y}$ may be specified at one $x_\mathrm{s}$, others at one (or more) other value(s) of $x$
   - Not all components may be specified
   - Some may be specified at multiple values of $x$
   - Gotta set a few, guess, poke around a bit. . .

# Main types of boundary conditions (recap)

1. $\vec{y}(x_s) = y_{\text{start}}$
   - Function and all derivatives are defined at some $x_s$
   - All that remains is to evolve the system forwards and/or backwards from $x_s$ to get $\vec{y}(x)$ for all $x$ of interest
   - This is an **Initial Value Problem** – covered today
2. Some components of $\vec{y}$ may be specified at one $x_s$, others at one (or more) other value(s) of $x$
   - Not all components may be specified
   - Some may be specified at multiple values of $x$
   - Gotta set a few, guess, poke around a bit…
   - Hrmmm, nasty – covered in Lecture 14.

# Main types of boundary conditions (recap)

1. $\vec{y}(x_\mathrm{s}) = y_\mathrm{start}$
   - Function and all derivatives are defined at some $x_\mathrm{s}$
   - All that remains is to evolve the system forwards and/or backwards from $x_\mathrm{s}$ to get $\vec{y}(x)$ for all $x$ of interest
   - This is an **Initial Value Problem** – covered today
2. Some components of $\vec{y}$ may be specified at one $x_\mathrm{s}$, others at one (or more) other value(s) of $x$
   - Not all components may be specified
   - Some may be specified at multiple values of $x$
   - Gotta set a few, guess, poke around a bit...
   - Hrmmm, nasty – covered in Lecture 14.
3. Some other more complicated auxiliary condition must be satisfied
   - e.g. $n_\mathrm{cookies}(\mathrm{today}) > n_\mathrm{critical}$ AND $n_\mathrm{cookies}(\mathrm{yesterday}) - n_\mathrm{cookies}(\mathrm{today}) < \mathit{MaxDailyConsumption}$

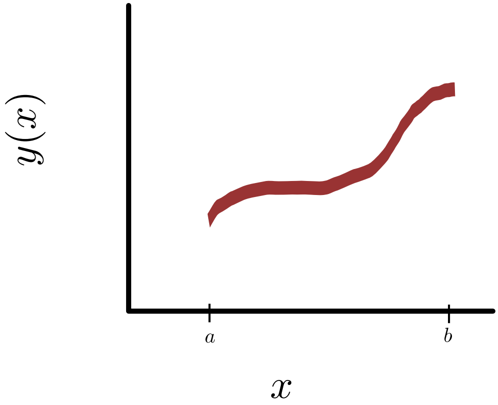
These last two are examples of **Boundary Value Problems**

# Outline

# Outline

# Euler's method (recap)

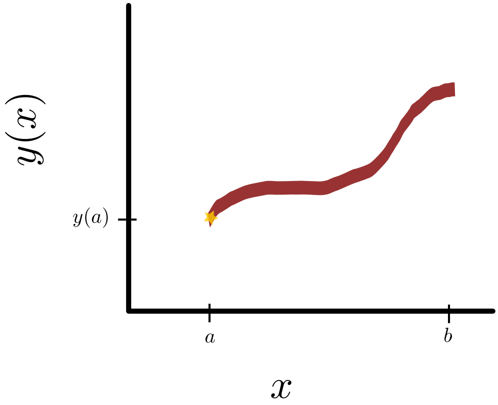
Extrapolating point-to-point using the derivative



# Euler's method (recap)

Extrapolating point-to-point using the derivative

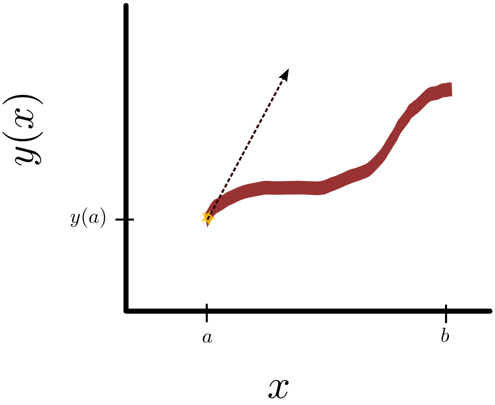
1. start with initial value $y(a)$



# Euler's method (recap)

Extrapolating point-to-point using the derivative

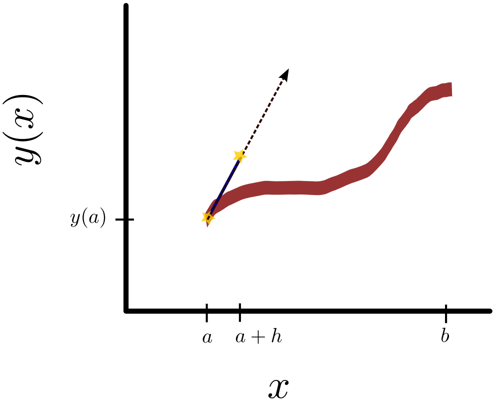
1. start with initial value $y(a)$
2. calculate $y'(a)$



# Euler's method (recap)

Extrapolating point-to-point using the derivative

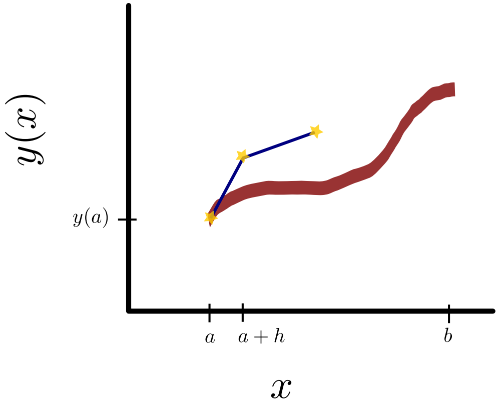
1. start with initial value $y(a)$
2. calculate $y'(a)$
3. step a distance $h$ in $x$, using $y'(a)$ to extrapolate behaviour of $y$



# Euler's method (recap)

Extrapolating point-to-point using the derivative

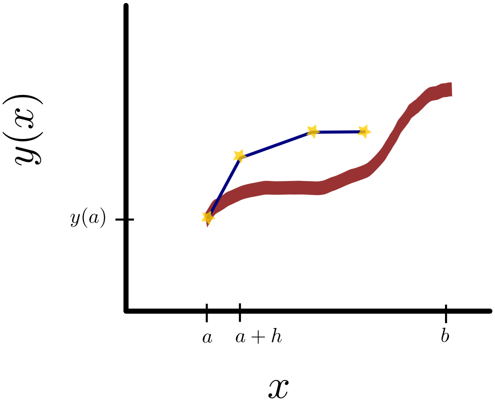
1. start with initial value $y(a)$
2. calculate $y'(a)$
3. step a distance $h$ in $x$, using $y'(a)$ to extrapolate behaviour of $y$
4. repeat until all interesting $x$ have been covered



# Euler's method (recap)

Extrapolating point-to-point using the derivative

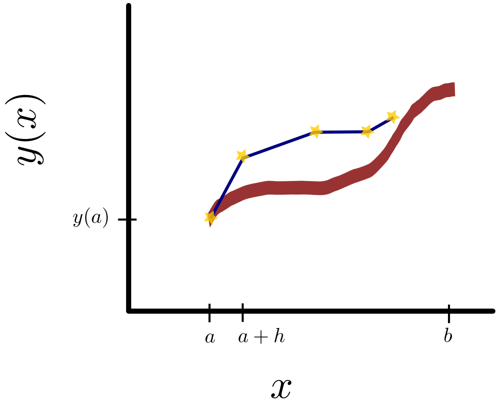
1. start with initial value $y(a)$
2. calculate $y'(a)$
3. step a distance $h$ in $x$, using $y'(a)$ to extrapolate behaviour of $y$
4. repeat until all interesting $x$ have been covered



# Euler's method (recap)

Extrapolating point-to-point using the derivative

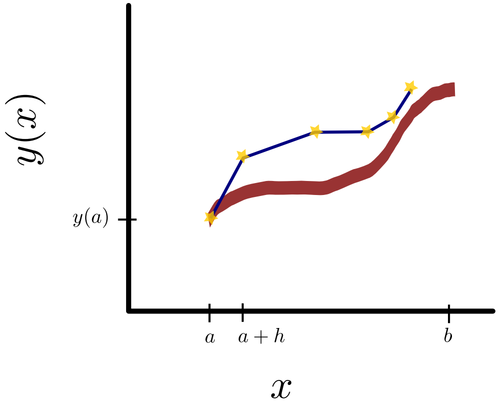
1. start with initial value $y(a)$
2. calculate $y'(a)$
3. step a distance $h$ in $x$, using $y'(a)$ to extrapolate behaviour of $y$
4. repeat until all interesting $x$ have been covered



# Euler's method (recap)

Extrapolating point-to-point using the derivative

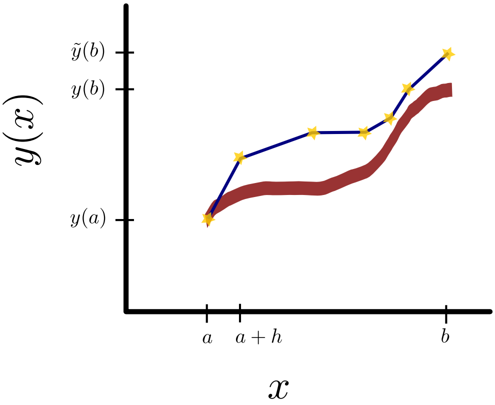
1. start with initial value $y(a)$
2. calculate $y'(a)$
3. step a distance $h$ in $x$, using $y'(a)$ to extrapolate behaviour of $y$
4. repeat until all interesting $x$ have been covered



# Euler's method (recap)

Extrapolating point-to-point using the derivative

1. start with initial value $y(a)$
2. calculate $y'(a)$
3. step a distance $h$ in $x$, using $y'(a)$ to extrapolate behaviour of $y$
4. repeat until all interesting $x$ have been covered
5. gives $\tilde{y}(b)$ as final estimate of $y(b)$



# Euler's method (recap)

Some things to remember about Euler's method

- Local error is $\mathcal{O}(h^2)$, global is $\mathcal{O}(h)$
- Requires just 1 evaluation of $f(x, y)$
- Asymmetric – uses derivative at start of interval
- $\implies$ essentially unstable (more on this later)
- Nonetheless, forms the conceptual basis of most ODE solvers

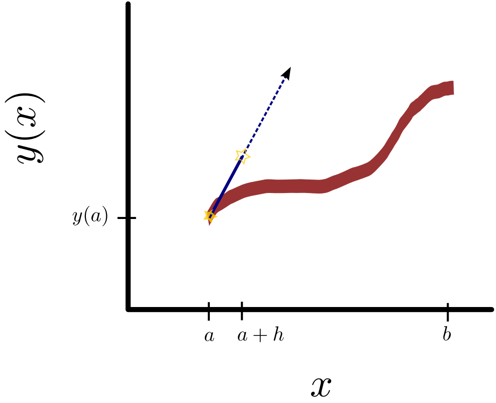
# The midpoint method

As usual, next refinement is to add an additional point
$\rightarrow$ midpoint $a + \frac{1}{2}h$

# The midpoint method

As usual, next refinement is to add an additional point $\rightarrow$ midpoint $a + \frac{1}{2}h$

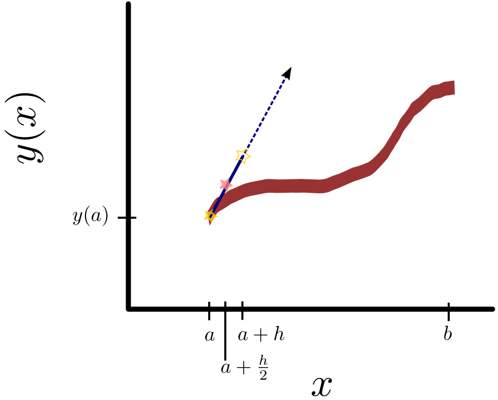
1. Evaluate $k_1 = hy'(a, y(a))$



# The midpoint method

As usual, next refinement is to add an additional point $\rightarrow$ midpoint $a + \frac{1}{2}h$

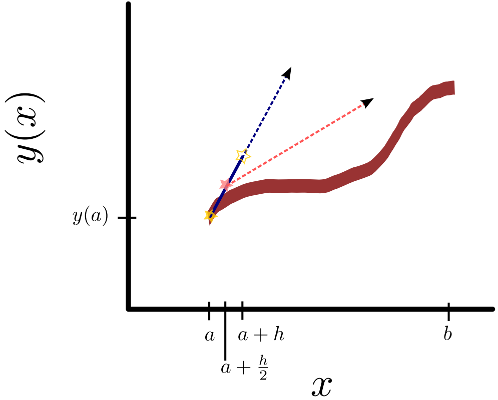
1. Evaluate $k_1 = hy'(a, y(a))$
2. Use this to extrapolate to $\tilde{y}(a + \frac{h}{2}) = y(a) + \frac{k_1}{2}$



# The midpoint method

As usual, next refinement is to add an additional point $\rightarrow$ midpoint $a + \frac{1}{2}h$

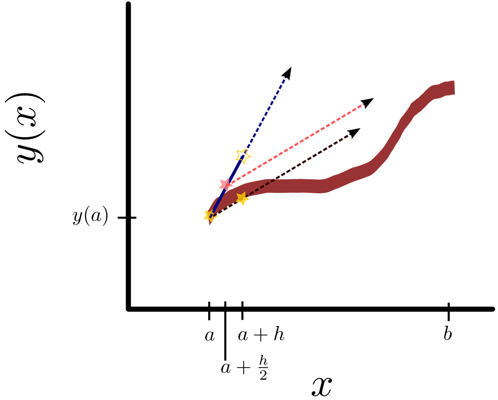
1. Evaluate $k_1 = hy'(a, y(a))$
2. Use this to extrapolate to $\tilde{y}(a + \frac{h}{2}) = y(a) + \frac{k_1}{2}$
3. Evaluate $k_2 = hy'(a + \frac{h}{2}, \tilde{y}(a + \frac{h}{2}))$



# The midpoint method

As usual, next refinement is to add an additional point $\rightarrow$ midpoint $a + \frac{1}{2}h$

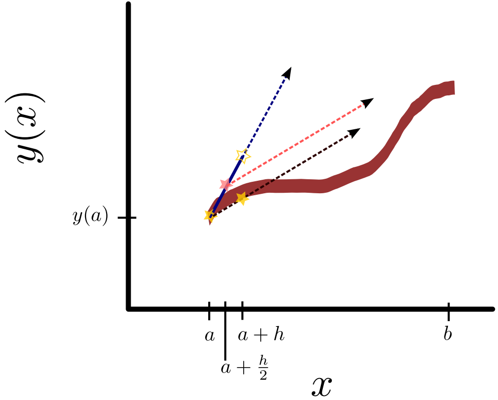
1. Evaluate $k_1 = hy'(a, y(a))$
2. Use this to extrapolate to $\tilde{y}(a + \frac{h}{2}) = y(a) + \frac{k_1}{2}$
3. Evaluate $k_2 = hy'(a + \frac{h}{2}, \tilde{y}(a + \frac{h}{2}))$
4. Use $k_2$ to extrapolate from $y(a)$ to $\tilde{y}(a + h)$



# The midpoint method

As usual, next refinement is to add an additional point $\rightarrow$ midpoint $a + \frac{1}{2}h$

1. Evaluate $k_1 = hy'(a, y(a))$
2. Use this to extrapolate to $\tilde{y}(a + \frac{h}{2}) = y(a) + \frac{k_1}{2}$
3. Evaluate $k_2 = hy'(a + \frac{h}{2}, \tilde{y}(a + \frac{h}{2}))$
4. Use $k_2$ to extrapolate from $y(a)$ to $\tilde{y}(a + h)$
5. Repeat to find each $\tilde{y}(x_{n+1})$ from each $\tilde{y}(x_n)$



# Midpoint (RK2) method

- This is a **2nd order Runge Kutta** (RK2) method: (dropping tildes)

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \\ y_{n+1} &= y_n + k_2 \end{aligned} \tag{1}$$

- Using the midpoint derivative actually cancels local errors of order $h^2$
  $\rightarrow$ local error is $\mathcal{O}(h^3)$, global is $\mathcal{O}(h^2)$
- Symmetric $\implies$ more stable than plain Euler's method
- Takes twice as many evaluations of $f$ as Euler's for same $h$
- $\implies$ Euler's *can* be faster occasionally, but not usually
- This may be the case if e.g. $y$ is not very smooth in $x$ and/or $y'$ does not depend on $y$

# Predictor-Corrector (RK2) method

- A related RK2 method is the **Predictor-Corrector**:

$$
\begin{aligned}
k_1 &= hf(x_n, y_n) \\
k_2 &= hf(x_n + \cancel{\tfrac{1}{2}}h, y_n + \cancel{\tfrac{1}{2}}k_1) \\
y_{n+1} &= y_n + \frac{1}{2}k_1 + \frac{1}{2}k_2
\end{aligned}
\tag{2}
$$

- Extra point also cancels local errors of order $h^2$
  $\rightarrow$ local error is $\mathcal{O}(h^3)$, global is $\mathcal{O}(h^2)$
- Overall method is therefore same order as midpoint method (RK$n$ = order $n$ global error)

# Leapfrog method

- Everything we have seen so far is a **single-step** method
- Can also make use of previous steps $\rightarrow$ **multi-step methods**

# Leapfrog method

- Everything we have seen so far is a **single-step** method
- Can also make use of previous steps $\rightarrow$ **multi-step methods**
- Simplest of these is the **leapfrog method**:

$$y(x+h) = y(x-h) + 2f(x, y(x))h \tag{3}$$

# Leapfrog method

- Everything we have seen so far is a **single-step** method
- Can also make use of previous steps $\rightarrow$ **multi-step methods**
- Simplest of these is the **leapfrog method**:

$$y(x+h) = y(x-h) + 2f(x, y(x))h \tag{3}$$

- Uses two points at a time $\rightarrow$ order $h^2$ global, for no more computations than Euler Method
- More efficient than e.g. predictor-corrector (half as many calls to $f$)
- Requires storage of previous steps (+ a way to start!)

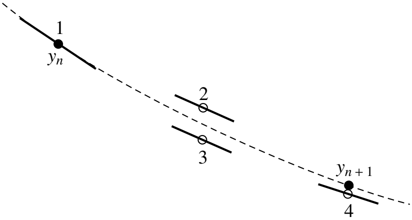
# Outline

# Fourth Order Runge-Kutta

OK, so let's stop messing around.

Weighted average of 4 different intermediate points
$\rightarrow$ weighting is such that $h^3$ and $h^4$ errors cancel
$\rightarrow$ error of order $h^5$, in just 4 evaluations of $f$

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\ k_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\ k_4 &= hf(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 \end{aligned} \tag{4}$$



# Reflections on RK4

- RK4 is a good, simple workhorse step for ODE integration
- With $y$ dependence collapsed, also very efficient for doing definite integrals (more next lecture)
- Can be made very competitive with decent adaptive timestepping

## Short break: feedback and planning

# Planning

- Mini-SOLE (informal feedback just to Yoshi & me) will be available in next Weds lab
- Projects are up on blackboard now:
  - Project 1: A Log Likelihood fit for extracting the *D*-meson lifetime
    **Key Topic: Functional Minimisation**
  - Project 2: Solving quantum systems numerically
    **Key Topic: Numerical integration and Monte Carlo methods**
  - Project 3: The Dynamics of Solitons
    **Key Topic: PDE Initial Value Problem**
  - Project 4: Heat dissipation in microprocessors
    **Key Topic: PDE solving with Neumann boundary conditions**
- Next lecture (Tues) we'll take a poll to get an idea of how many people expect to do each
- The demonstrators will get assigned a 'primary project' and 'secondary project' to focus on.

# Outline

# Using Error Estimates: Timestep Rescaling

- We know local error from RK4 and other such 4th order RK-like formulae is $\propto h^5$
- Therefore, error $\Delta_1$ observed with some stepsize $h_1$ compares to desired error $\Delta_0$ (with optimal timestep $h_0$) as

$$\frac{\Delta_1}{\Delta_0} = \left(\frac{h_1}{h_0}\right)^5 \qquad \Longrightarrow \quad h_0 = h_1 \left(\frac{\Delta_0}{\Delta_1}\right)^{\frac{1}{5}} \tag{5}$$

- Provides automatic contraction *or* expansion of $h$ to achieve largest $h$ that returns desired accuracy
- Whether you set $\Delta_0 = \epsilon$ or $\Delta_0 = \epsilon y$ depends on whether you want to specify relative or absolute accuracy
- $\Delta_0$ refers to a vector of errors when working with $n$ coupled ODEs – generally just use lowest common denominator (largest error)

# Making Error Estimates: Step Doubling

- Take each step twice: once as a full step, once as 2 half-steps
- Leads to two estimates of $y_{n+1}$:

$$y_{n+1}^{(1-\text{step})} = y_1 + (2h)^5 \times (\textit{some constant}) + \mathcal{O}(h^6) \tag{6}$$
$$y_{n+1}^{(2-\text{step})} = y_2 + 2h^5 \times (\textit{some constant}) + \mathcal{O}(h^6) \tag{7}$$

- Can estimate error $\Delta_1$ to 4th order as

$$\Delta_1 = y_2 - y_1 \tag{8}$$

- Can also subtract Eq. 6 from Eq. 7 and rearrange to get

$$y_{n+1}^{(5)} = y_2 + \frac{\Delta_1}{15} + \mathcal{O}(h^6), \tag{9}$$

$\rightarrow$ improved estimate of $y_{n+1}$, good to 5th order

# Making Error Estimates: Embedded RK formulae

Possible to write higher-order RK formulae with lower-order formulae embedded – e.g. 6-point, 5th-order step

$$\begin{aligned}
k_1 &= hf(x_n, y_n) \\
k_2 &= hf(x_n + a_2 h, y_n + b_{21} k_1) \\
&\dots \\
k_6 &= hf(x_n + a_6 h, y_n + b_{61} k_1 + \dots + b_{65} k_5)
\end{aligned} \tag{10}$$

$$y_{n+1}^{(5)} = y_n + \sum_{i=1}^{6} c_i k_i$$

which contains an embedded 4th-order step

$$y_{n+1}^{(4)} = y_n + \sum_{i=1}^{6} c_i^{\mathrm{alt}} k_i \tag{11}$$

(all constants can be found in Numerical Recipes: 'Cash-Karp' constants)

# Making Error Estimates: Embedded RK formulae

- Use 5th order scheme $y_{n+1}^{(5)}$ as RK5 step
- Use difference between $y_{n+1}^{(5)}$ and $y_{n+1}^{(4)}$ as 4th order estimate
$$\Delta_1 = y_{n+1}^{(5)} - y_{n+1}^{(4)} \tag{12}$$
in
$$h_0 = h_1 \left(\frac{\Delta_0}{\Delta_1}\right)^{\frac{1}{5}}$$
- About a factor of 2 quicker than step doubling
- Both this and step doubling known as ‘RK45’ in the business (5th order result, 4th order error estimate)
- Long-time standard techniques for ODEs and definite integrals

# A Note: Making Timestep Rescaling Safe

$$\frac{\Delta_1}{\Delta_0} = \left(\frac{h_1}{h_0}\right)^5 \qquad \Longrightarrow \quad h_0 = h_1 \left(\frac{\Delta_0}{\Delta_1}\right)^{\frac{1}{5}} \tag{13}$$

works, but extrapolates linearly, which may not be so safe...

Better to incorporate some limits and a safety factor $S$, e.g.

$$h_0 = h_1 S \left(\frac{\Delta_0}{\Delta_1}\right)^{\frac{1}{5}} \tag{14}$$

with $\mathcal{O}(1 - S) \sim 10^{-2}$, and restrict $h_0/h_1$ such that

$$0.2 \lesssim \frac{h_0}{h_1} \lesssim 10 \tag{15}$$

# Outline

# A bit of an aside...

Euler's method is unstable as a forward = explicit scheme

$$y_{n+1} = y_n + hy'(x_n, y_n) \tag{16}$$

# A bit of an aside. . .

Euler's method is unstable as a forward = explicit scheme

$$y_{n+1} = y_n + hy'(x_n, y_n) \tag{16}$$

Can also be cast as a backwards = implicit scheme

$$y_{n+1} = y_n + hy'(x_{n+1}, y_{n+1}) \tag{17}$$

# A bit of an aside...

Euler's method is unstable as a forward = explicit scheme

$$y_{n+1} = y_n + hy'(x_n, y_n) \tag{16}$$

Can also be cast as a backwards = implicit scheme

$$y_{n+1} = y_n + hy'(x_{n+1}, y_{n+1}) \tag{17}$$

- Implicit equation for $y_{n+1}$ at each step

# A bit of an aside...

Euler's method is unstable as a forward = explicit scheme

$$y_{n+1} = y_n + hy'(x_n, y_n) \tag{16}$$

Can also be cast as a backwards = implicit scheme

$$y_{n+1} = y_n + hy'(x_{n+1}, y_{n+1}) \tag{17}$$

- Implicit equation for $y_{n+1}$ at each step
- Less accurate

# A bit of an aside…

Euler's method is unstable as a forward = explicit scheme

$$y_{n+1} = y_n + hy'(x_n, y_n) \tag{16}$$

Can also be cast as a backwards = implicit scheme

$$y_{n+1} = y_n + hy'(x_{n+1}, y_{n+1}) \tag{17}$$

- Implicit equation for $y_{n+1}$ at each step
- Less accurate
- Completely stable

# A bit of an aside. . .

Euler's method is unstable as a forward = explicit scheme

$$y_{n+1} = y_n + hy'(x_n, y_n) \tag{16}$$

Can also be cast as a backwards = implicit scheme

$$y_{n+1} = y_n + hy'(x_{n+1}, y_{n+1}) \tag{17}$$

- Implicit equation for $y_{n+1}$ at each step
- Less accurate
- Completely stable
- Complete pain in the ass to solve.

# A bit of an aside...

Euler's method is unstable as a forward = explicit scheme

$$y_{n+1} = y_n + hy'(x_n, y_n) \tag{16}$$

Can also be cast as a backwards = implicit scheme

$$y_{n+1} = y_n + hy'(x_{n+1}, y_{n+1}) \tag{17}$$

- Implicit equation for $y_{n+1}$ at each step
- Less accurate
- Completely stable
- Complete pain in the ass to solve.

All methods today were explicit – but all have implicit versions.

- Usually involve doing some expensive matrix inversion at each step
- Useful for **stiff systems** (multiscale coupled ODEs)

# Housekeeping

- Assignment due 12 noon on Monday
- Tuesday: Numerical Integration (Yoshi)