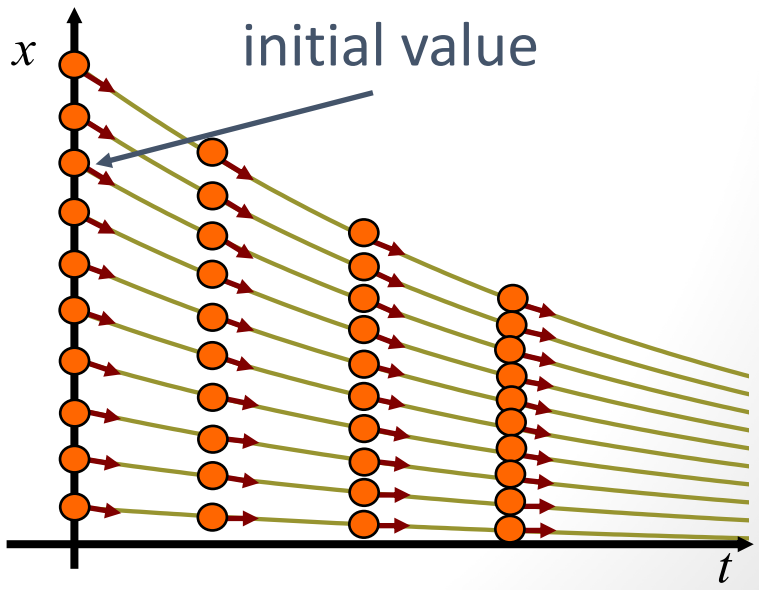
ODE helper

Monday, 14 October 2019

Initial Value Problems

- Typically, the additional condition to uniquely determine the solution is the initial value of the function at $t=t_0$
- For a n^{th}-order ODE (that can be converted to the system of n 1^{st}-order ODEs) we need n initial conditions (at *one* point, i.e. for x, dx/dt, d^2x/dt^2,… at t_0)
- IVP problem: find $x(t)$ such that

$$\frac{dx}{dt} = f(x(t),t), \quad t>0$$
$$x(0) = x_0$$



Euler forward method

- In the expression for the ODE: $\frac{dx}{dt} = f(x(t),t), \quad x(0) = x_0$
- At the time t_k approximate *dx/dt* with the forward difference:

$$\frac{dx}{dt} = \frac{x(t_{k+1}) - x(t_k)}{h} + O(h), \quad h = t_{k+1} - t_k$$

therefore, we approximate values of $x(t_k)$

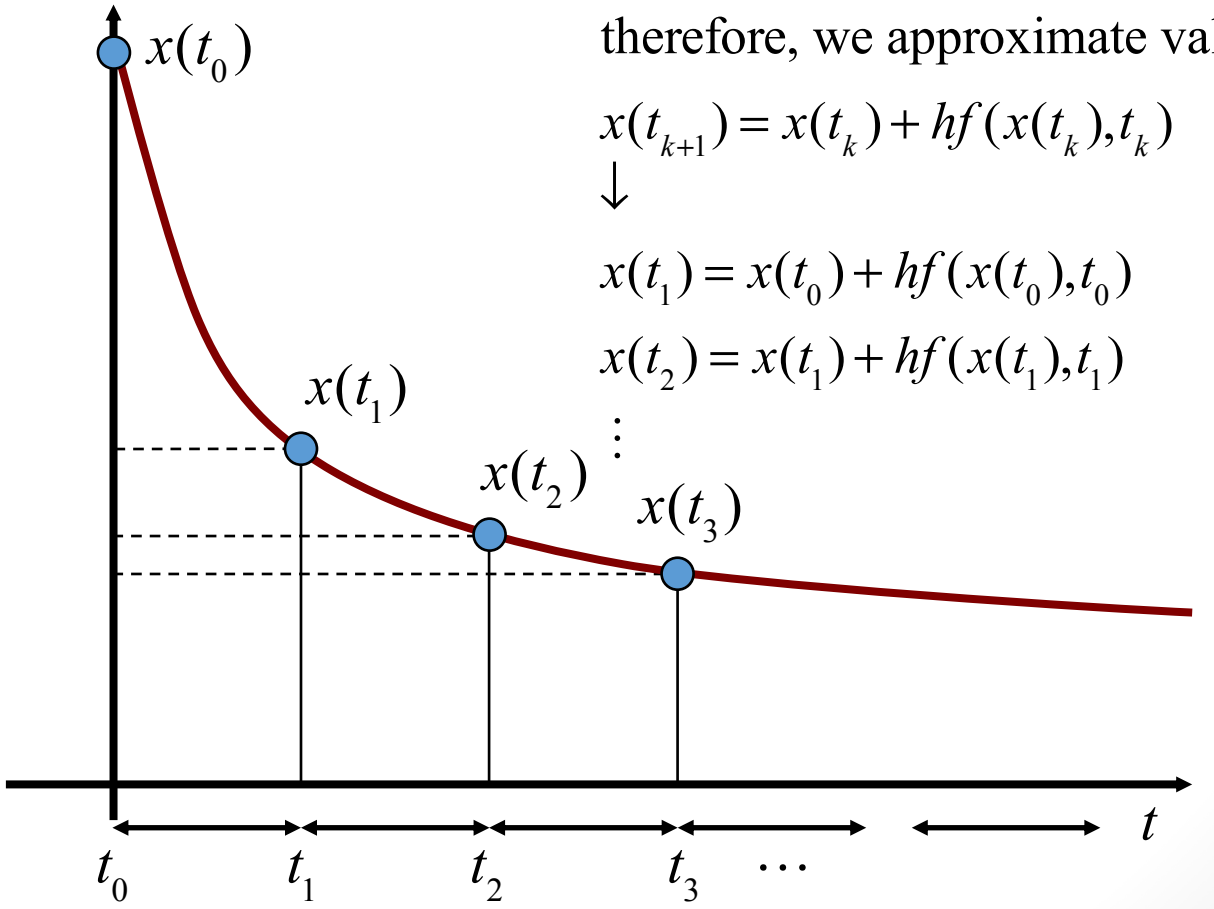
$$x(t_{k+1}) = x(t_k) + hf(x(t_k),t_k)$$

$$\downarrow$$

$$x(t_1) = x(t_0) + hf(x(t_0),t_0)$$

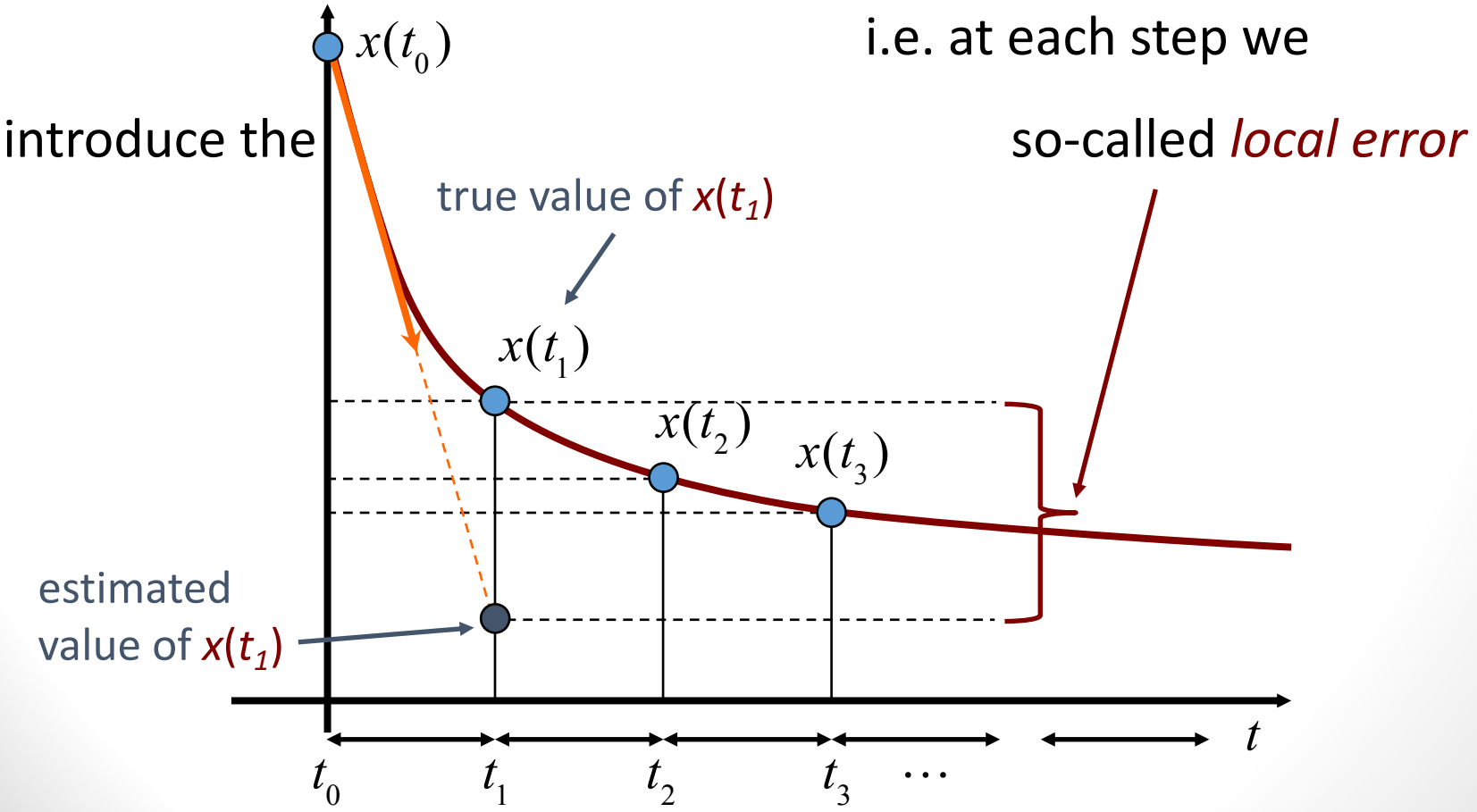
$$x(t_2) = x(t_1) + hf(x(t_1),t_1)$$

$$\vdots$$



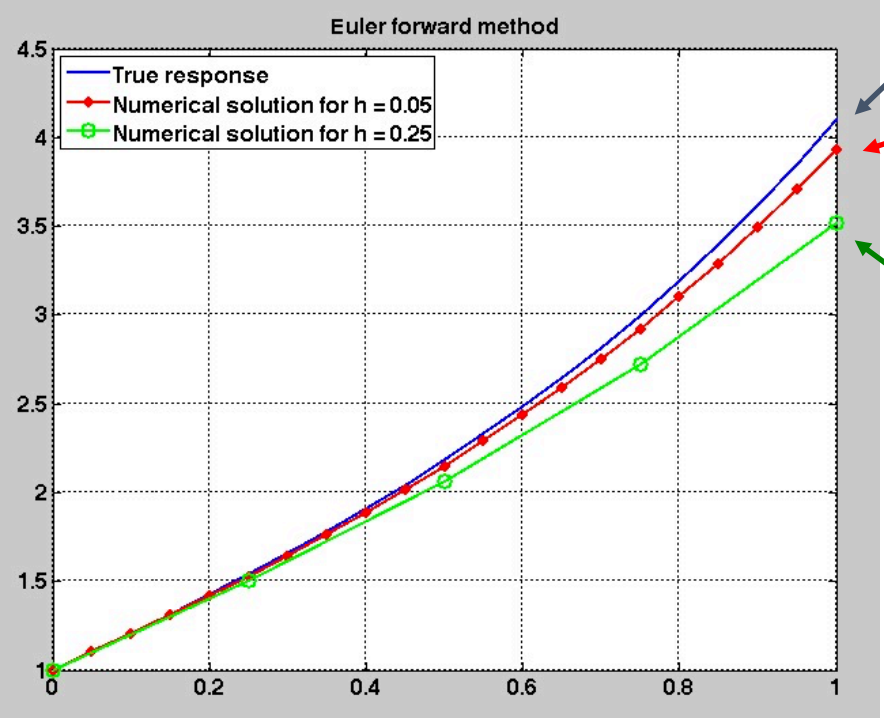
Euler forward method

- This method is *single-step* and *explicit*, i.e. uses information at t_k to compute the solution at t_{k+1}
- The slope at t_k is estimated using forward difference



Euler forward method

- Example: $\frac{dx}{dt} = 2x - 3t, \quad x(0) = 1$
- We use the Euler forward method: $x(t_{k+1}) = x(t_k) + hf(x(t_k), t_k)$



true value of $x(t)$

computed value of $x(t)$ for h=0.05

computed value of $x(t)$ for h=0.25

Error analysis in Euler methods

- Use the Taylor series to expand around *x(t)*

$$x(t_{k+1}) = x(t_k) + hf(x(t_k), t_k) + \frac{h^2}{2} f'(x(t_k), t_k) + \dots$$

Euler forward method — local error

- *Local error* is of $O(h^2)$ – if we halve *h* then the local error will reduce by factor of four
- However, if we halve *h* then we need twice more steps to cover the desired interval -> we introduce twice as more *local errors*
- At the end of the interval, *global error* halves as we halve *h*
- Therefore, these methods are first-order methods (proportional to *h*)

Higher-order Taylor series methods

- For the ODE:

$$\frac{dx}{dt} = f(x(t),t), \quad x(0) = x_0$$

- Taylor expansion:

$$x(t_{k+1}) = x(t_k) + hf(x(t_k),t_k) + \frac{h^2}{2} f'(x(t_k),t_k) + \ldots + \frac{h^n}{n!} f^{(n-1)}(x(t_k),t_k) + O(h^{n+1})$$

Euler forward method with $O(h^2)$

2^{nd}-order Taylor series method with $O(h^3)$

- Have to estimate $f'(x(t_k),t_k)$, $f''(x(t_k),t_k)$,… i.e. as the order increases, it becomes complicated to compute derivatives
- These methods are not frequently used

Runge-Kutta methods

- Motivation: improve the accuracy of solution without calculating higher order derivatives
- General formulation:

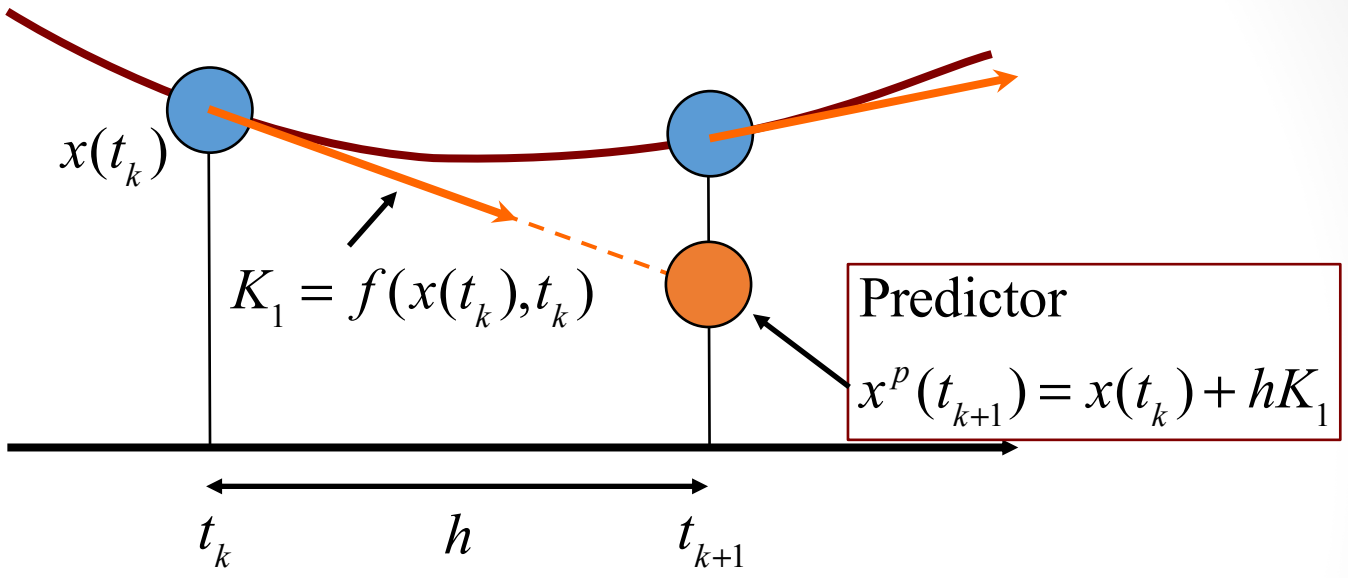
$$x(t_{k+1}) = x(t_k) + h(a_1 K_1 + a_2 K_2 + \cdots + a_n K_n)$$

$$K_1 = f(x(t_k), t_k)$$
$$K_2 = f(x(t_k) + h\beta_{1,1} K_1, t_k + h\alpha_1)$$
$$\vdots$$
$$K_n = f(x(t_k) + h\beta_{n-1,1} K_1 + h\beta_{n-1,2} K_2 + \cdots + h\beta_{n-1,n-1} K_{n-1}, t_k + h\alpha_{n-1})$$

Runge-Kutta (RK) of n^{th} order

- $a_1, \ldots, \alpha_1, \ldots, \beta_{1,1}, \ldots$ - constants derived in such a way that the approximation matches as many terms in the Taylor expansion as possible

Geometric interpretation of 2nd–order Runge-Kutta method

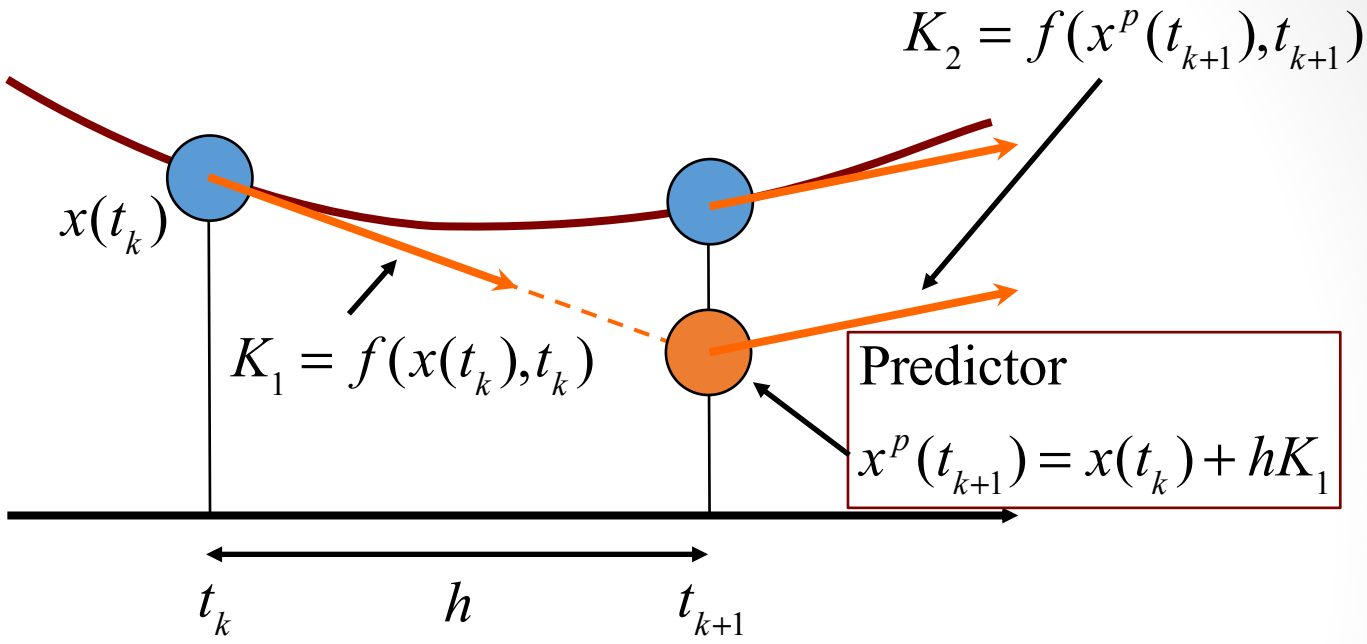


$$x(t_{k+1}) = x(t_k) + \frac{h}{2}(K_1 + K_2)$$

$$K_1 = f(x(t_k), t_k)$$

$$K_2 = f(\underbrace{x(t_k) + hK_1}_{x^p(t_{k+1})}, \underbrace{t_k + h}_{t_{k+1}})$$

Geometric interpretation of 2nd–order Runge-Kutta method

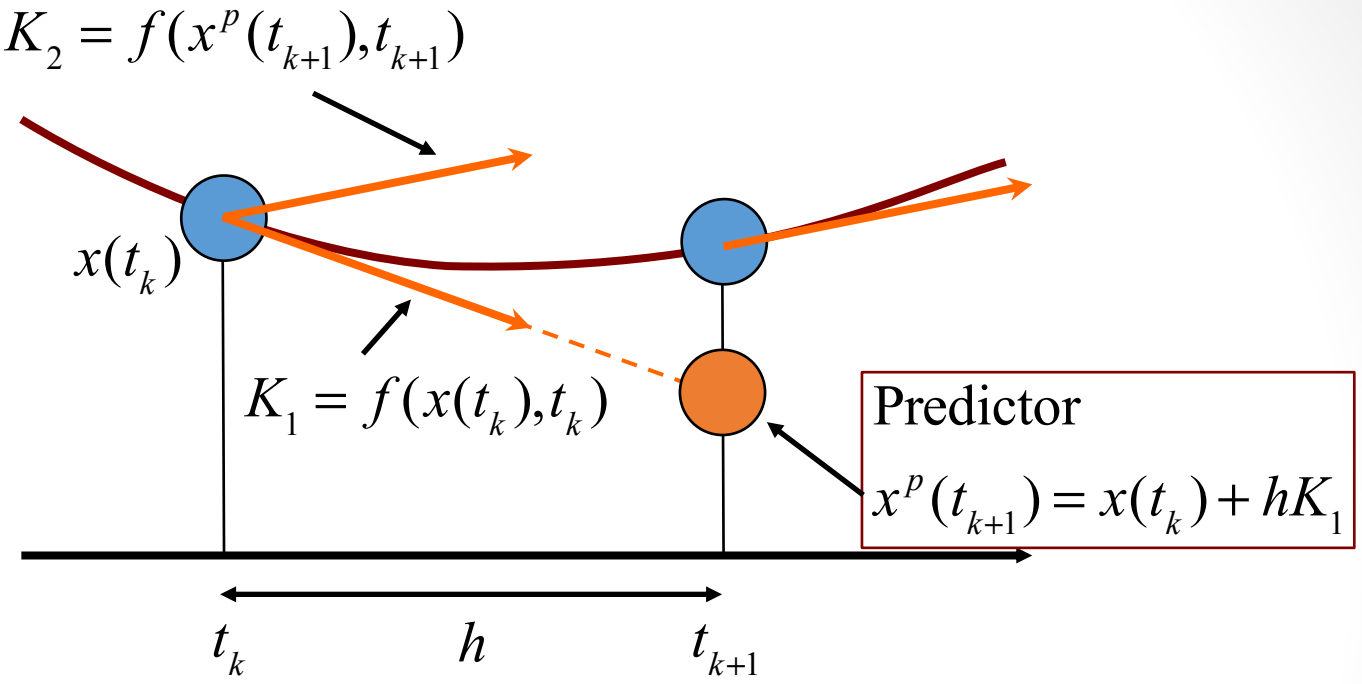


$$x(t_{k+1}) = x(t_k) + \frac{h}{2}(K_1 + K_2)$$

$$K_1 = f(x(t_k), t_k)$$

$$K_2 = f(\underbrace{x(t_k) + hK_1}_{x^p(t_{k+1})}, \underbrace{t_k + h}_{t_{k+1}})$$

Geometric interpretation of 2nd–order Runge-Kutta method

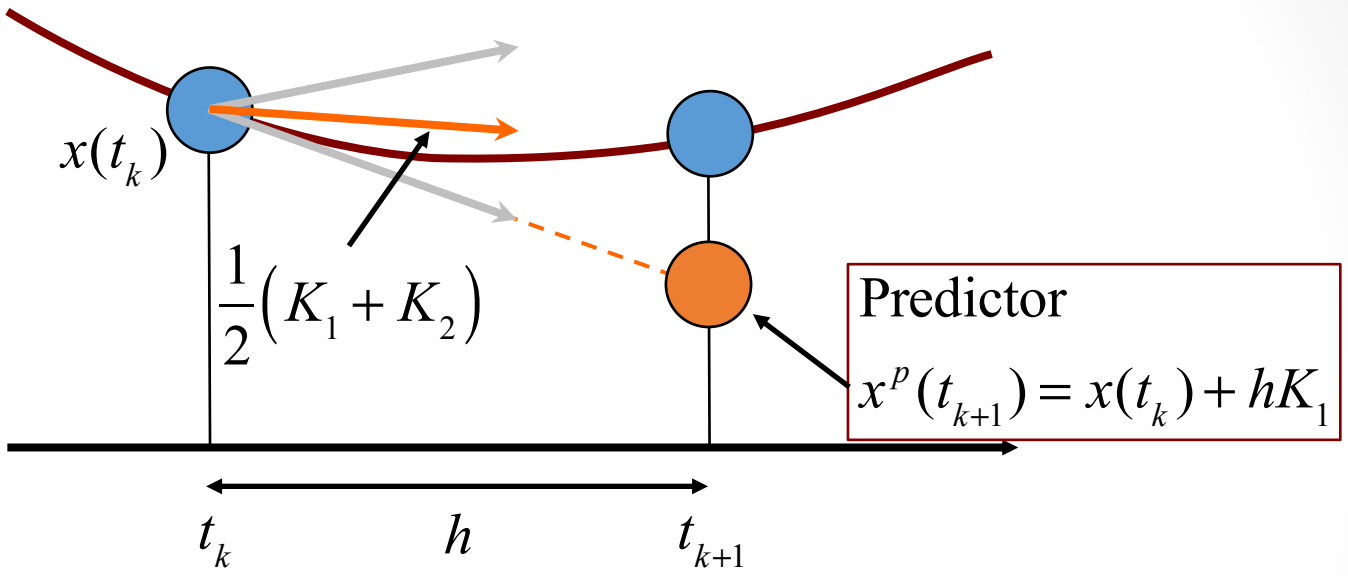


$$x(t_{k+1}) = x(t_k) + \frac{h}{2}(K_1 + K_2)$$

$$K_1 = f(x(t_k), t_k)$$

$$K_2 = f(\underbrace{x(t_k) + hK_1}_{x^p(t_{k+1})}, \underbrace{t_k + h}_{t_{k+1}})$$

Geometric interpretation of 2nd–order Runge-Kutta method

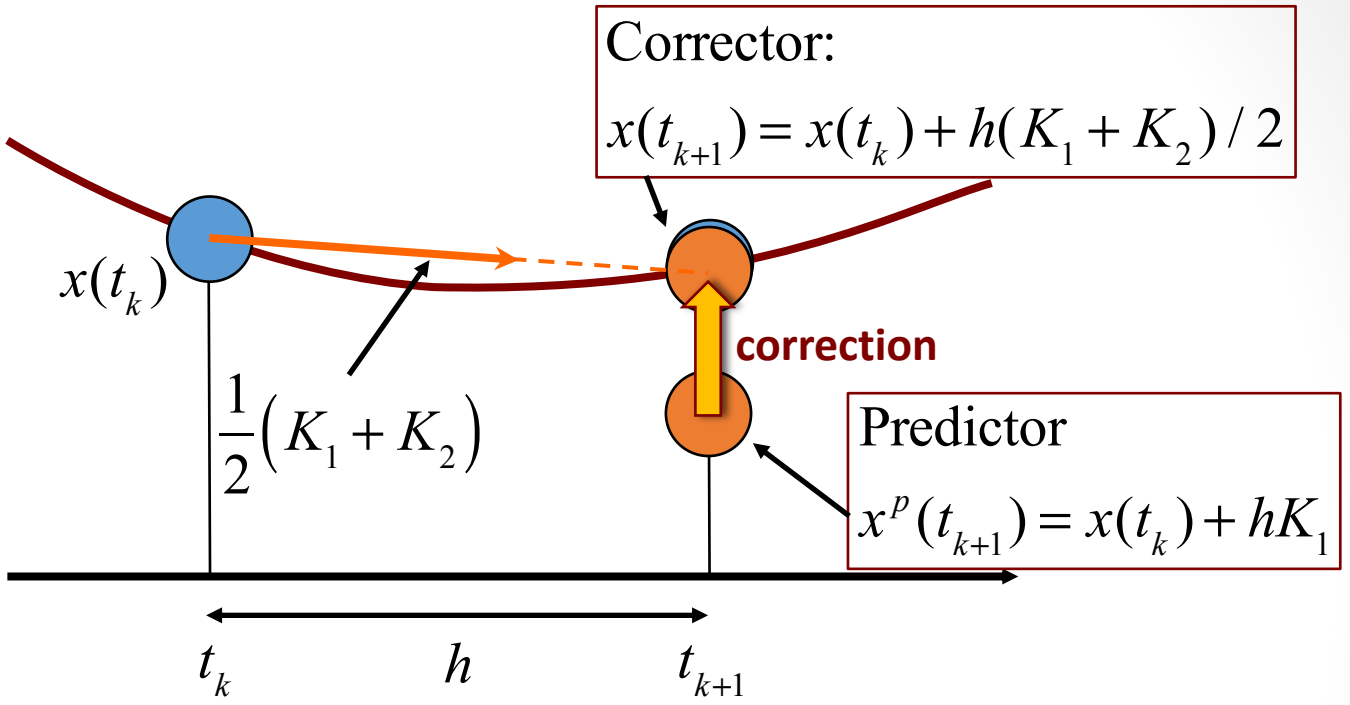


$$x(t_{k+1}) = x(t_k) + \frac{h}{2}(K_1 + K_2)$$

$$K_1 = f(x(t_k), t_k)$$

$$K_2 = f(\underbrace{x(t_k) + hK_1}_{x^p(t_{k+1})}, \underbrace{t_k + h}_{t_{k+1}})$$

Geometric interpretation of 2nd–order Runge-Kutta method



$$x(t_{k+1}) = x(t_k) + \frac{h}{2}(K_1 + K_2)$$
$$K_1 = f(x(t_k), t_k)$$
$$K_2 = f(\underbrace{x(t_k) + hK_1}_{x^p(t_{k+1})}, \underbrace{t_k + h}_{t_{k+1}})$$

Predictor

$$x^p(t_{k+1}) = x(t_k) + hK_1$$

Corrector:

$$x(t_{k+1}) = x(t_k) + \frac{h}{2}(K_1 + f(x^p(t_{k+1}), t_{k+1}))$$

4th–order Runge-Kutta method

$$x(t_{k+1}) = x(t_k) + \frac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4)$$

$$K_1 = f(x(t_k), t_k)$$

$$K_2 = f(x(t_k) + \frac{h}{2}K_1, t_k + \frac{h}{2})$$

$$K_3 = f(x(t_k) + \frac{h}{2}K_2, t_k + \frac{h}{2})$$

$$K_4 = f(x(t_k) + hK_3, t_k + h)$$

- 4th-order RK have the local truncation error of $O(h^5)$
- Four function evaluations needed
- RK methods require no solutions prior to time t_k
  - as a consequence, it is easy to change the step size during integration
  - it is a self-starting method

Stiff systems

- Stiff ODEs describe systems with very different time scales, i.e. some components of these systems evolve relatively slowly whereas others are changing rapidly
- The Jacobian matrix of this kind of systems has eigenvalues that differ greatly in magnitude
- Euler forward method is very inefficient in solving stiff systems as stability can be ensured only with very small steps (the rapidly varying component, i.e. large λ, calls for a small h)
- Example:

*INTEGRATION OF STIFF EQUATIONS**

By C. F. Curtiss and J. O. Hirschfelder

The Naval Research Laboratory, Department of Chemistry, University of Wisconsin, Madison, Wisconsin

Communicated by Farrington Daniels, December 29, 1951

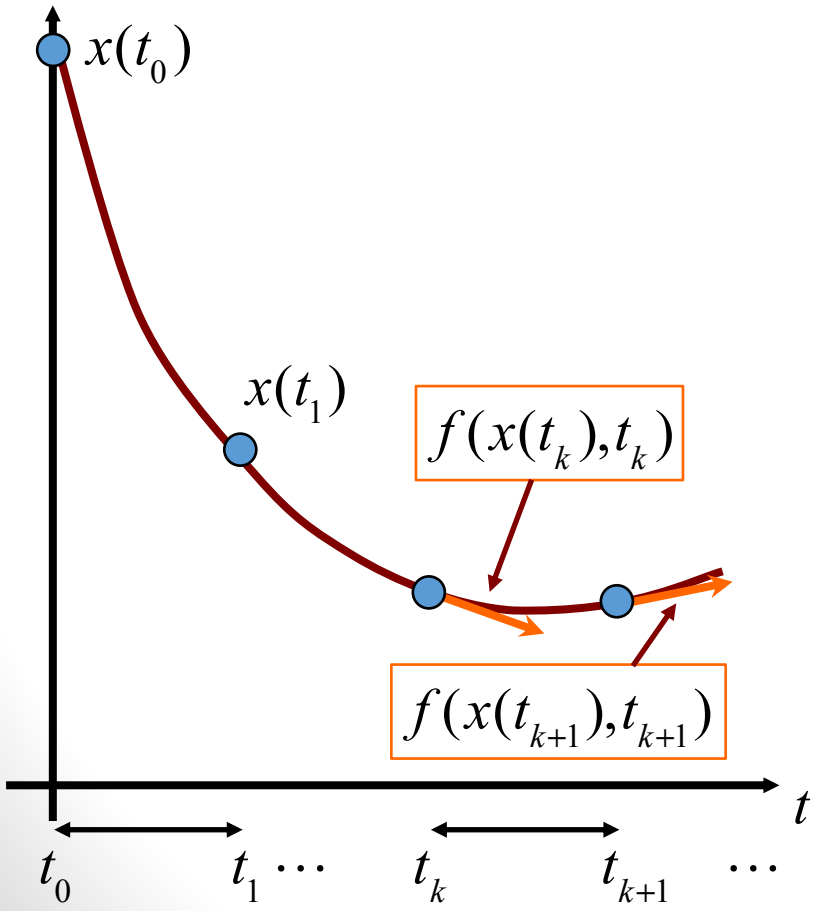
In the study of chemical kinetics, electrical circuit theory, and problems of missile guidance a type of differential equation arises which is exceedingly difficult to solve by ordinary numerical procedures. A very satisfactory method of solution of these equations is obtained by making use of a forward interpolation process. This scheme has the unusual property

Euler backward method

- At t_k approximate *dx/dt* with the backward finite difference:

$$\frac{dx}{dt} = f(x(t),t), \quad x(0) = x_0 \quad \Rightarrow \quad \frac{x(t_{k+1}) - x(t_k)}{h} = f(x(t_{k+1}),t_{k+1}) + O(h)$$

$$h = t_{k+1} - t_k$$



therefore, neglecting $O(h)$, we have

$$x(t_{k+1}) - x(t_k) - hf(x(t_{k+1}),t_{k+1}) = 0$$

since $x(t_{k+1})$ is unknown, we have to resolve the nonlinear system, where we take $w=x(t_{k+1})$:

$$g(w) = w - x(t_k) - hf(w,t_{k+1})$$

For example, use Newton-Raphson

$$w_{n+1} = w_n - \frac{w_n - x(t_k) - hf(w_n,t_{k+1})}{1 - h\frac{\partial f}{\partial w}(w_n,t_{k+1})}, \; n = 0,1,\ldots$$

Initial condition:

(i) $w_0 = x(t_k)$

(ii) Euler forward difference estimate of $x(t_{k+1})$

Euler backward method

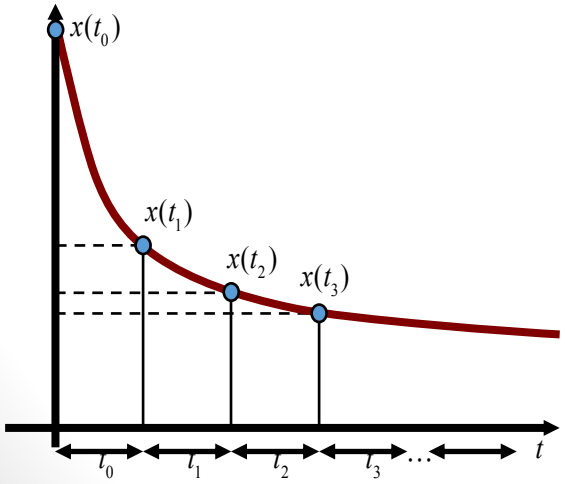
- Example: $\frac{dx}{dt} = 2x - 3t, \quad x(0) = 1$
- In each step we have to solve:

$$x(t_{k+1}) - x(t_k) - hf(x(t_{k+1}), t_{k+1}) = 0$$

$$w - x(t_k) - h \cdot (2 \cdot w - 3 \cdot t_{k+1}) = 0$$

$$w = \frac{x(t_k) - 3 \cdot h \cdot t_{k+1}}{(1 - 2h)}$$

- For $h = t_{k+1} - t_k = 0.1$ we have:



$$x(t_2 = 0.1) = w = \frac{1 - 3 \cdot 0.1 \cdot 0.1}{1 - 2 \cdot 0.1} = 1.2125$$

$$x(t_2 = 0.2) = w = \frac{1.2125 - 0.3 \cdot 0.2}{0.8} = 1.4406$$

$$x(t_3 = 0.3) = w = \frac{1.4406 - 0.3 \cdot 0.3}{0.8} = 1.6883$$

$$\vdots$$

$$x(t_{10} = 1) = w = 4.5783$$

Explicit vs Implicit methods

- Mentioned previously: Euler forward method is *explicit*, i.e. f is evaluated with x_k at time t_k to compute the solution $x_{k+1}(t_{k+1})$
- Another alternative: evaluate f with x_{k+1} before we know its value (at time t_{k+1}). Methods with this feature are *implicit*
- Implicit methods necessitate more computations as it is required to solve algebraic equations to compute x_{k+1}
- Implicit methods are more robust (i.e. have larger stability region than explicit methods)
- Therefore, implicit methods are more appropriate for solving stiff systems

ODE solvers in python

The solvers are implemented as individual classes which can be used directly (low-level usage) or through a convenience function.

solve_ivp(fun, t_span, y0[, method, t_eval, ...])	Solve an initial value problem for a system of ODEs.
RK23(fun, t0, y0, t_bound[, max_step, rtol, ...])	Explicit Runge-Kutta method of order 3(2).
RK45(fun, t0, y0, t_bound[, max_step, rtol, ...])	Explicit Runge-Kutta method of order 5(4).
Radau(fun, t0, y0, t_bound[, max_step, ...])	Implicit Runge-Kutta method of Radau IIA family of order 5.
BDF(fun, t0, y0, t_bound[, max_step, rtol, ...])	Implicit method based on backward-differentiation formulas.
LSODA(fun, t0, y0, t_bound[, first_step, ...])	Adams/BDF method with automatic stiffness detection and switching.
OdeSolver(fun, t0, y0, t_bound, vectorized)	Base class for ODE solvers.
DenseOutput(t_old, t)	Base class for local interpolant over step made by an ODE solver.
OdeSolution(ts, interpolants)	Continuous ODE solution.

ODE solvers in Matlab

Solver	Problem Type	Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. `ode45` should be the first solver you try.
`ode23`		Low	`ode23` can be more efficient than `ode45` at problems with crude tolerances, or in the presence of moderate stiffness.
`ode113`		Low to High	`ode113` can be more efficient than `ode45` at problems with stringent error tolerances, or when the ODE function is expensive to evaluate.
`ode15s`	Stiff	Low to Medium	Try `ode15s` when `ode45` fails or is inefficient and you suspect that the problem is stiff. Also use `ode15s` when solving differential algebraic equations (DAEs).
`ode23s`		Low	`ode23s` can be more efficient than `ode15s` at problems with crude error tolerances. It can solve some stiff problems for which `ode15s` is not effective. `ode23s` computes the Jacobian in each step, so it is beneficial to provide the Jacobian via `odeset` to maximize efficiency and accuracy. If there is a mass matrix, it must be constant.
`ode23t`		Low	Use `ode23t` if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve differential algebraic equations (DAEs).
`ode23tb`		Low	Like `ode23s`, the `ode23tb` solver might be more efficient than `ode15s` at problems with crude error tolerances.
`ode15i`	Fully implicit	Low	Use `ode15i` for fully implicit problems $f(t,y,y') = 0$ and for differential algebraic equations (DAEs) of index 1.