

# Unit 1: Review

**ODEs** are differential equations with one single independent variable (1D), & the functions derivatives depend only on this independent variable

$$\frac{d^n f(x)}{dx^n} = F(x, f(x), f'(x), \dots, f^{(n-1)}(x))$$

**IVP** are problems where the initial conditions are known, e.g., for a  $n^{\text{th}}$  order diff.eq., the initial values for the  $0^{\text{th}}$  to  $(n-1)^{\text{th}}$  derivatives are known!!

$$x=0 \Rightarrow f(x=0), f'(x=0), \dots, f^{(n-1)}(x=0) \text{ known}$$

Using Sympy (Steps)

- 1) Import sympy
- 2) Define variable symbols with `sp.Symbol('x')`
- 3) Define the function with `sp.Function('f')(x)`
- 4) Define the derivatives with `sp.diff(f, x, m)` or  $f.\underset{\substack{\text{mth order}}}{\text{diff}}(x, m)$
- 5) Define differential equation `sp.Eq(lhs, rhs)`
- 6) Solve with `sp.dsolve(eq, f, ics = {f(0) = f0})`  
 $\uparrow$   
initial conditions

For a system of equations:

- 5.1) Define set of ODE's `eqs = (sp.Eq(...), sp.Eq(..), ...)` as a tuple or list

6.1) Solve with `sp.dsolve(eq)` or `sp.dsolve-system(eq)`

To check solution:

7) Import `checkodesol` and `checkodesol(eq, f[i])`

## The Euler Methods

$$\frac{dS(t)}{dt} = \overbrace{F(t, S(t))}^{\text{slope}},$$

Using linear approximation (forward diff), in a regular discretized grid, we obtain

$$\frac{dS(t_i)}{dt} = \frac{S(t_{i+1}) - S(t_i)}{(t_{i+1} - t_i)}$$

Explicit Euler Formula  $\rightarrow S(t_{j+1}) = S(t_j) + h \frac{dS(t_j)}{dt} = S(t_j) + h \underbrace{F(t_j, S(t_j))}_{\text{slope}}$  1st order method! Tends to overestimate

Using the initial value  $S(t_0)$  we obtain the next  $S(t_j)$  (time evolution) approximation by applying the formula.

- Store  $S(t_0)$  in an empty array
- Compute  $S(t_1) = S(t_0) + h F(t_0, S(t_0))$  & store it
- Compute  $S(t_2) = S(t_1) + h F(t_1, S(t_1))$  & store it
- $\vdots$
- Compute  $S(t_f) = S(t_{f-1}) + h F(t_{f-1}, S(t_{f-1}))$  & store it
- $S$  is the approximation!

Using a backward difference,

$$\text{The Implicit Euler Formula} \rightarrow S(t_{j+1}) = S(t_j) + h F(t_{j+1}, S(t_{j+1}))$$

1<sup>st</sup> Order

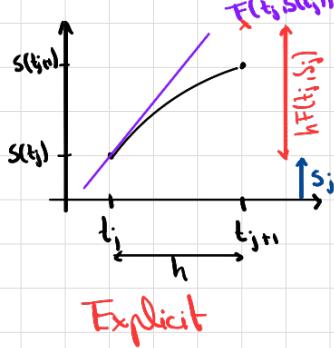
Tends to underestimate

And using an average of them, we obtain

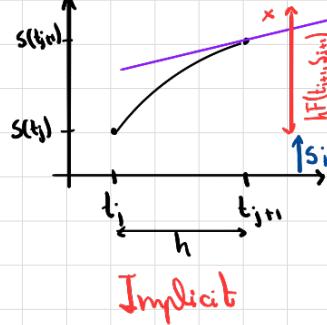
$$\text{The Trapezoidal rule} \rightarrow S(t_{j+1}) = S(t_j) + \frac{h}{2} [F(t_j, S(t_j)) + F(t_{j+1}, S(t_{j+1}))]$$

2<sup>nd</sup> Order

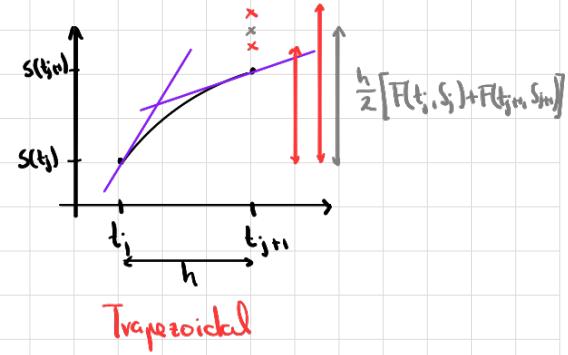
can be consider as a 2<sup>nd</sup> order Runge Kutta or multi-step method



Explicit



Implicit



Trapezoidal

Slopes

$$\frac{df(x)}{dx} = \underbrace{F(x, f(x))}_{\text{slope}}$$

Now, if the slope depends explicitly on  $f(x)$  we need a PREDICTOR statement!!

for implicit methods

Usually, we do

- 1)  $f(x_{i+1}) = f(x_i) + h F(x_i, f(x_i))$
- 2)  $f(x_{i+1}) = f(x_i) + h F(x_{i+1}, f(x_{i+1}))$

## Reduction of Order

Many numerical methods are designed specifically for 1<sup>st</sup> order diff. equations, that's why we should turn any possible problem into a 1<sup>st</sup> order problem!

So, it's useful to define a state vector  $S(t)$ , such that

$$\vec{S}(t) = \begin{bmatrix} f(t) \\ f''(t) \\ \vdots \\ f^{(n-1)}(t) \end{bmatrix} \Rightarrow \frac{d\vec{S}(t)}{dt} = \begin{bmatrix} f'(t) \\ \vdots \\ f^{(n)}(t) \end{bmatrix} = \begin{bmatrix} S_1(t) \\ \vdots \\ S_n(t) \\ F(t, S_1, \dots, S_n) \end{bmatrix} = \vec{F}(t, \vec{S}(t))$$

as the initial problem is

$$\frac{d^n f(t)}{dt^n} = F(t, f(t), \dots, f^{(n-1)}(t))$$

Then, we have an equivalent problem

$$\frac{d\vec{S}(t)}{dt} = \vec{F}(t, \vec{S}(t))$$

Example: S. Pendulum  $\ddot{\theta}(t) = -\frac{g}{l} \theta(t)$

Let's define,

$$\vec{S}(t) = \begin{bmatrix} \theta(t) \\ \dot{\theta}(t) \end{bmatrix} \Rightarrow \frac{d\vec{S}}{dt} = \begin{bmatrix} \dot{\theta}(t) \\ \ddot{\theta}(t) \end{bmatrix} = \begin{bmatrix} S_1(t) \\ -\frac{g}{l} S_1(t) \end{bmatrix}$$

given in the ODE

And we can write it in matrix form as

$$\frac{d\vec{S}}{dt} = \underbrace{\begin{bmatrix} 0 & 1 \\ \frac{g}{l} & 0 \end{bmatrix}}_{\vec{F}(t, \vec{S}(t))} \vec{S}$$

Then, to use our methods, we do

- Explicit  $\vec{S}(t_{j+1}) = \vec{S}(t_j) + h \vec{F}(t_j, \vec{S}(t_j)) = I \vec{S}(t_j) + \begin{bmatrix} 0 & h \\ \frac{g}{l}h & 0 \end{bmatrix} \vec{S}(t_j)$   
 $= \begin{bmatrix} 1 & h \\ \frac{g}{l}h & 1 \end{bmatrix} \vec{S}(t_j)$

- Implicit  $\vec{S}(t_{j+1}) = \vec{S}(t_j) + h \vec{F}(t_{j+1}, \vec{S}(t_{j+1}))$   
 $= \vec{S}(t_j) + \begin{bmatrix} 0 & h \\ \frac{g}{l}h & 0 \end{bmatrix} \vec{S}(t_{j+1}) \Rightarrow \vec{S}(t_{j+1}) = \underbrace{\begin{bmatrix} 1 & -h \\ \frac{g}{l}h & 1 \end{bmatrix}^{-1}}_{+} \vec{S}(t_j)$

- Trapezoidal  $\vec{S}(t_{j+1}) = \vec{S}(t_j) + \frac{h}{2} \left[ \vec{F}(t_j, \vec{S}(t_j)) + \vec{F}(t_{j+1}, \vec{S}(t_{j+1})) \right]$

$$\vec{S}(t_{j+1}) = \underbrace{\begin{bmatrix} 1 & -\frac{h}{2} \\ \frac{g}{l}h & 1 \end{bmatrix}^{-1}}_{+} \underbrace{\begin{bmatrix} 1 & \frac{h}{2} \\ \frac{g}{l}h & 1 \end{bmatrix}}_{+} \vec{S}(t_j)$$

Notice we have no implicit dependency anymore!  
 That's why we don't need a predictor step !!

• **Accuracy** refers to the scheme's ability to get close to the exact solution, usually unknown, as a function of the step size, i.e.,  $O(h^2)$

• **Stability** refers to the scheme's ability to keep the error from growing as it integrates forward in time. If the error grows, unstable; otherwise is stable.

Euler Methods are unstable !! Thus we need more methods to get over this stability problem!

## Predictor - Corrector Methods

This methods rely on two main stages/steps :

1) Predictor statement is an explicit estimation of the solution at  $t_{j+1}$  (We could use Euler Explicit)

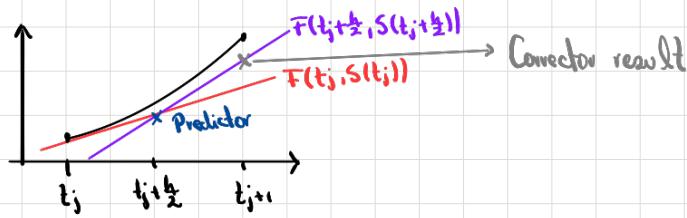
2) Correction statement uses the previous estimation to calculate a new more accurate solution

Using these steps we aim to improve the approximation by querying  $F$  (predictor) at different locations and then use an average of the predictions ('correction') to update the state

Example: Mid-Point method

$$1) \text{ Predictor: } S(t_{j+1}) = S(t_j) + \frac{h}{2} F(t_j, S(t_j))$$

$$2) \text{ Corrector: } S(t_{j+1}) = S(t_j) + h F\left(t_j + \frac{h}{2}, S(t_j + \frac{h}{2})\right)$$



## Runge Kutta Methods

We can improve Euler methods by keeping higher order terms in the Taylor expansion for  $S(t_j + h)$  around  $t_j$

$$S(t_{j+1}) = S(t_j + h) = S(t_j) + S'(t_j) \underbrace{(t_j + h - t_j)}_h + \frac{1}{2} S''(t_j) h^2 + \dots + \frac{1}{n!} S^{(n)}(t_j) h^n$$

## Second Order Runge Kutta Method

Taking the expansion up to the 2<sup>nd</sup> order, we get

$$\begin{aligned} S(t_{j+1}) &= S(t_j) + h S'(t_j) + \frac{1}{2} h^2 S''(t_j) + O(h^3) \quad \text{, this method has a 2nd} \\ &= S(t_j) + h F(t_j, S(t_j)) + \frac{1}{2} h^2 \frac{dF(t_j, S(t_j))}{dt} \quad (I) \quad \text{order of accuracy} \end{aligned}$$

$$\text{but, } \frac{dF(t_j, S(t_j))}{dt} = \frac{\partial F}{\partial t} + \frac{\partial F}{\partial S} \frac{dS}{dt} = \frac{\partial F}{\partial t} + \frac{\partial F}{\partial S} F$$

Now, we want to develop a method with 2 slopes, such that

$$\cdot k_1 = F(t_j, S(t_j)) \quad \cdot k_2 = F\left(t_j + ph, S(t_j) + qh k_1\right)$$

So,

$$S(t_{j+1}) = S(t_j) + h \underbrace{(c_1 k_1 + c_2 k_2)}_{\text{average slope}}$$

fraction of step      weighted slope

Expanding  $k_2$ , we obtain

$$k_2 = F(t_j, S(t_j)) + ph \frac{\partial F}{\partial t} + qh \frac{\partial F}{\partial S} F$$

and then,

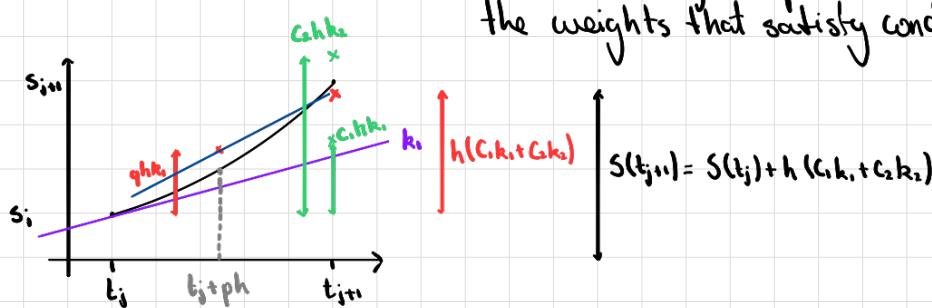
$$\begin{aligned} S(t_{j+1}) &= S(t_j) + h \left[ C_1 F + C_2 \left( F + ph \frac{\partial F}{\partial t} + qh \frac{\partial F}{\partial S} F \right) \right] \\ &= S(t_j) + (C_1 + C_2) h F + \left( p \frac{\partial F}{\partial t} + q \frac{\partial F}{\partial S} F \right) C_2 h \quad (\text{II}) \end{aligned}$$

Comparing (I) & (II):

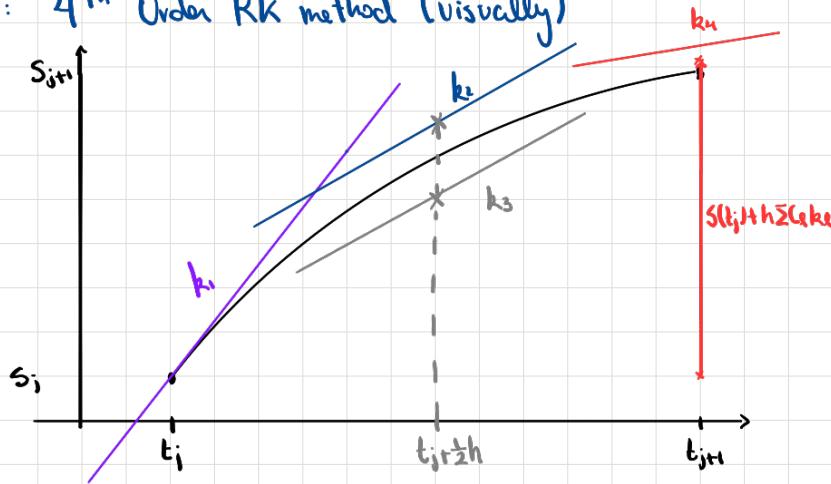
$$\begin{array}{l} \cdot C_1 + C_2 = 1 \\ \cdot C_2 p = \frac{1}{2} \\ \cdot C_2 q = \frac{1}{2} \end{array} \left. \begin{array}{l} \text{conditions} \\ \text{for} \\ 2^{\text{nd}} \text{ order method} \end{array} \right.$$

In general, for  $n^{\text{th}}$  order method:

- 1)  $n$  slopes, each depending on the previous one
- 2) Taylor expansion
- 3) Conditions by comparison & assign values for the weights that satisfy conditions



Example: 4<sup>th</sup> Order RK method (visually)



Here we assume  $C_1 = \frac{1}{6} = C_2$ ,  $C_3 = C_4 = \frac{1}{3}$   
 $C_2 p_1 = \frac{1}{2} = C_2 q_1$ ,  $C_3 p_2 = C_3 q_2 = \frac{1}{2}$

We need 4 slopes, such that  
 $S(t_{i+1}) = S(t_i) + h \sum_{k=1}^4 C_k k_k$

Now,  $k_1$  is given by ODE, then  $k_2$  is obtained with  $k_1$  and evaluated in some point inside  $t_i + ph$ ,  $k_3$  is calculated using  $k_1$  &  $k_2$  and evaluated at  $t_i + \sqrt{h}$ ,  $k_4$  is calculated using  $k_1$ ,  $k_2$  &  $k_3$  and evaluated at  $t_{i+1}$ .

Remember: Specific choice of parameters can reduce the order of the method!

## Scipy ODE solvers for IVP

Using `scipy.integrate.solve_ivp()`, we need to linearize the problem and provide the initial value condition.

$$\frac{dS}{dt} = F(t, S(t))$$

- 1) Define the "function"  $F$  (slope)
- 2) Define the initial condition  $S(0)$
- 3) Use `solve_ivp(F, t_span, S(0), method='RK4', t_eval=None)`  
↓  
int. time interval

just if we want to evaluate solution in some  $t$  interval

## Systems of ODE

First, we linearize the system by defining a state vector containing the independent variables. Then we do the same previous process

## ODE Boundary Problems

In this type of problems we know the values of the function at the extremes of the independent variable!

Shooting Method is a 3 stages method : 1) Aim      2) Shoot      3) Compare  
(guess)      (construct path)      (Final check & repeat)

This method turn the problem into a initial value problem!

For a 2<sup>nd</sup> order ODE, 1) Guess  $f'(x=a)$  & together with the known  $f(x=a)$  we turn the problem into a IVP (Aim)

2) Using known methods we get an approximate solution (Shoot)  
Euler, RK, etc  
and get to  $f(x=b) = f_\beta$  (boundary)

3) Compare the given  $f_b$  & our obtained  $f_\beta$  and adjust a new guess. Repeat until "Convergence" (Compare & iterate)

So, finally this problem leads to a root finding problem!

$$f_\beta - f_b = 0 \text{ or } g(\alpha) - f_b = 0$$

$$\Rightarrow |S(t_f) - S_{\text{shoot}}| \leq \text{tolerance}$$

Example :  $\frac{d^2 f(x)}{dx^2} = -10$ , we need to reduce its order  $\Rightarrow \vec{S}(x) = \begin{bmatrix} f(x) \\ f'(x) \end{bmatrix}$ ,  $\frac{d\vec{S}}{dx} = \begin{bmatrix} f'(x) \\ f''(x) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -10 & 0 \end{bmatrix} \vec{S}(x) +$   
 $f(0) = 500, f'(0) = 100$

Now, a good procedure is to try to obtain 2 bounds for the guess. So, let's try two times.

- Define a guess for  $f(0)$  & then we get  $\vec{S}(0) = \begin{bmatrix} f(0) \\ f'(0) \end{bmatrix}$
- Use RK4 method to construct  $\vec{S}(t)$
- Compare  $\vec{S}_{\text{shoot}}(x=20) = f_{\text{shoot}}(x=20)$  with  $f(20)$ , change guess & repeat.

Then, we use optimization to find the root of  $S_{\text{shoot}}(x=20) - f(x=20) = 0$ ,

→ 1) Define obj\_fun(v\_guess):

```
sol = solve_ivp(slope, [x_00, x_20], np.asarray(S(0)), dtypes='object'),
           method='RK45', t_eval=x_axis)
```

last\_f = sol.y[0][-1]

return last\_f - f\_20

2) Carry out optimization, new\_v00 = opt.fsolve(obj\_fun, v\_00)

3) Get  $\vec{S}(t)$  using this optimized guess new\_v00

## ODE Finite Difference Method for Boundary Problems

This method turns the ODE into a system of algebraic equations ( $n+1$  equations)

Using central differences:

$$\frac{dy}{dx} = \frac{y_{i+1} - y_{i-1}}{2h}$$

$$\frac{d^2y}{dx^2} = \frac{y_{i-1} - 2y_i + y_{i+1}}{h^2}$$

if we divide the grid in  $n$  equal subintervals

Example:

$$\frac{df}{dx} = -10 \Rightarrow f(0) = 500 \quad f(20) = 100$$

$$\text{for } i\text{th: } \frac{f_{i-1} - 2f_i + f_{i+1}}{h^2} = -10 \Rightarrow f_{i-1} - 2f_i + f_{i+1} = -10h^2$$

in the ODE

Now, we can express this in matrix form as

$$\left( \begin{array}{cccccc|c} 1 & 0 & \dots & \dots & 0 & f_0 = f(0) \\ 1 & -2 & 1 & 0 & 0 & f_1 \\ 0 & 1 & -2 & 1 & 0 & \vdots \\ 0 & 0 & 1 & -2 & 1 & \vdots \\ \vdots & \ddots & \ddots & 0 & 0 & f_{n-1} \\ 0 & \dots & \dots & \dots & 1 & f_n = f(20) \end{array} \right) = \left( \begin{array}{c} 500 \\ -10h^2 \\ -10h^2 \\ \vdots \\ -10h^2 \\ 100 \end{array} \right)$$

Finally we solve this system with any linear algebra package!  $Ax=b$   
 $x = A^{-1}b$ ,