Unit 1 : Review

Rolando Sánchez

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

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

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

$x = 0 \Rightarrow f(x=0),\ f''(x=0),\ \ldots,\ f^{(n-1)}(x=0)$ 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.diff(x, m)` (m: m^{th} order)
5) Define differential equation `sp.Eq(lhs, rhs)`
6) Solve with `sp.dsolve(eq, f, ics = {f(0): f0})` (ics: initial conditions)

For a system of equations:

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

6.1) Solve with `sp.dsolve(eqs)` or `sp.dsolve_system(eq)`

To check solution:

7) Import `checkodesol` and `checkodesol(eqs, 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

$$\frac{dS(t_j)}{dt} = \frac{S(t_{j+1}) - S(t_j)}{(t_{j+1} - t_j)}$$

Explicit Euler Formula $\longrightarrow$ $S(t_{j+1}) = S(t_j) + h\frac{dS(t_j)}{dt} = S(t_j) + h\underbrace{F(t_j, S(t_j))}_{\text{slope}}$

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 $S(t_1) = S(t_0) + hF(t_0, S(t_0))$ & store it
- Compute $S(t_2) = S(t_1) + hF(t_1, S(t_1))$ & store it
- ⋮
- Compute $S(t_f) = S(t_{f-1}) + hF(t_{f+1}, S(t_{f-1}))$ & store it
- S is the approximation!

Using a backward difference,

The Implicit Euler Formula → $S(t_{j+1}) = S(t_j) + hF(t_{j+1}, S(t_{j+1}))$

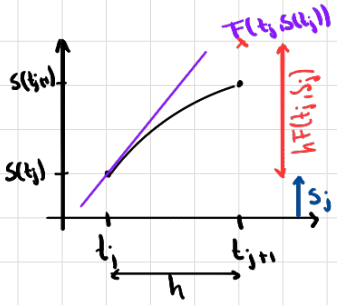
1st Order
Tends to underestimate

And using an average of them, we obtain

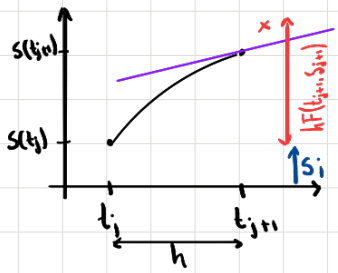
The Trapezoidal Rule → $S(t_{j+1}) = S(t_j) + \frac{h}{2}\left[F(t_j, S(t_j)) + F(t_{j+1}, S(t_{j+1}))\right]$

2nd Order

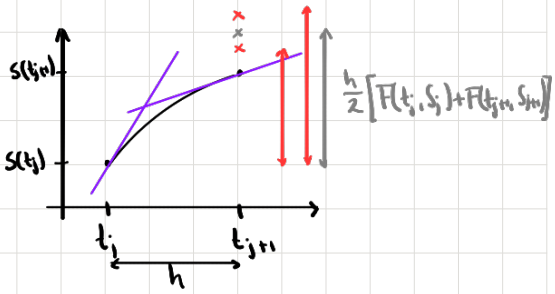
can be consider as a 2nd order Runge Kutta or multi-step method



Explicit



Implicit



Trapezoidal

Slopes

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

Now, if the slope depends explicitely on $f(x)$ we need a PREDICTOR statement!!
for implicit methods

Usually, we do
1) $f(x_{i+1}) = f(x_i) + hF(x_i, f(x_i))$
2) $f(x_{i+1}) = f(x_i) + hF(x_{i+1}, f(x_{i+1}))$

Reduction of Order

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

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

$$\vec{S}(t) = \begin{bmatrix} f(t) \\ f^{(1)}(t) \\ \vdots \\ f^{(n-1)}(t) \end{bmatrix} \Rightarrow \frac{d\vec{S}(t)}{dt} = \begin{bmatrix} f^{(1)}(t) \\ \vdots \\ f^{(n)}(t) \end{bmatrix} = \begin{bmatrix} S_2(t) \\ \vdots \\ S_n(t) \\ F(t, S_1, \dots, S_n) \end{bmatrix} = \vec{F}(t, \vec{S}(t))$$

as the initial problem is

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

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,

$$\vec{S}(t) = \begin{bmatrix} \theta(t) \\ \dot{\theta}(t) \end{bmatrix} \Rightarrow \frac{d\vec{S}}{dt} = \begin{bmatrix} \dot{\theta}(t) \\ \ddot{\theta}(t) \end{bmatrix} = \begin{bmatrix} S_2(t) \\ -\frac{g}{l}S_1(t) \end{bmatrix}$$

($\ddot{\theta}(t)$: given in the ODE)

And we can write it in matrix form as

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

Then, to use our methods, we do

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

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

- Trapezoidal $$\vec{S}(t_{j+1}) = \vec{S}(t_j) + \frac{h}{2}\left[\vec{F}(t_j, \vec{S}(t_j)) + \vec{F}(t_{j+1}, \vec{S}(t_{j+1}))\right]$$
$$\vec{S}(t_{j+1}) = \begin{bmatrix} 1 & -\frac{h}{2} \\ \frac{gh}{2l} & 1 \end{bmatrix}^{-1}\begin{bmatrix} 1 & \frac{h}{2} \\ -\frac{gh}{2l} & 1 \end{bmatrix}\vec{S}(t_j)$$

Notice we have no implicit dependency anymore! That's why we do not 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 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)

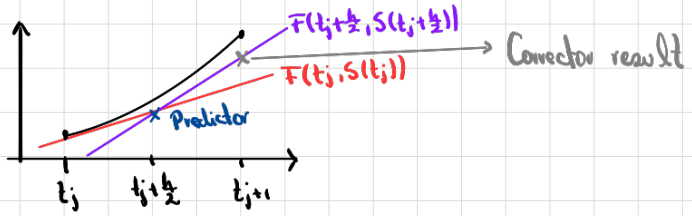
2) **Correction statement** uses the previous estimation to calculate a new more accurate solution

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

2) Corrector: $S(t_{j+1}) = S(t_j) + hF(t_j+\frac{h}{2}, S(t_j+\frac{h}{2}))$



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

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

Second Order Runge Kutta Method

Taking the expansion up to the 2nd order, we get

$$S(t_{j+1}) = S(t_j) + hS'(t_j) + \frac{1}{2}h^2 S''(t_j) + O(h^3)$$
$$= S(t_j) + hF(t_j, S(t_j)) + \frac{1}{2}h^2 \frac{dF(t_j, S(t_j))}{dt} \quad (I)$$

this method has a 2nd order of accuracy

but, $\frac{dF(t_j, S(t_j))}{dt} = \frac{\partial F}{\partial t} + \frac{\partial F}{\partial S}\frac{dS}{dt} = \frac{\partial F}{\partial t} + \frac{\partial F}{\partial S}F$ ← and we replace it here!

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

- $k_1 = F(t_j, S(t_j))$
- $k_2 = F(\underbrace{t_j + ph}_{\text{fraction of step}}, S(t_j) + \underbrace{qhk_1}_{\text{weighted slope}})$

So,

$$S(t_{j+1}) = S(t_j) + h(\underbrace{C_1k_1 + C_2k_2}_{\text{average slope}})$$

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,

$$S(t_{j+1}) = S(t_j) + h\left[C_1F + C_2\left(F + ph\frac{\partial F}{\partial t} + qh\frac{\partial F}{\partial S}F\right)\right]$$

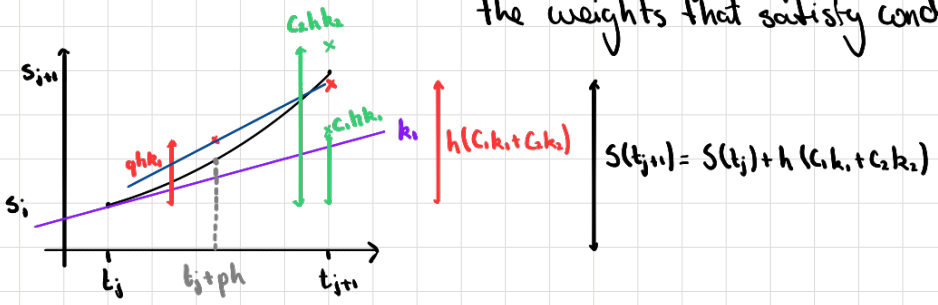
$$= S(t_j) + (C_1+C_2)hF + \left(p\frac{\partial F}{\partial t} + q\frac{\partial F}{\partial S}F\right)C_2h \quad (II)$$

Comparing (I) & (II):

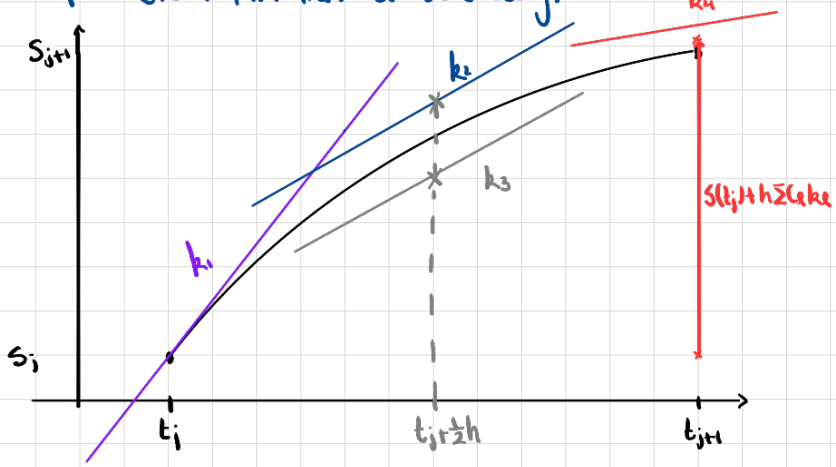
- $C_1 + C_2 = 1$
- $C_2 p = \frac{1}{2}$
- $C_2 q = \frac{1}{2}$

} Conditions for 2nd order method

In general, for n^{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: 4th Order RK method (visually)



We need 4 slopes, such that

$$S(t_{j+1}) = S(t_j) + h\sum_{l=1}^{4} C_l k_l$$

Now, k_1 is given by ODE, then k_2 is obtained with k_1 and evaluated in some point inside $t_j + ph$, k_3 is calculated using k_1 & k_2 and evaluated at $t_j + \frac{1}{2}h$, k_4 is calculated using k_1, k_2 & k_3 and evaluated at t_{j+1}.

Here we assumed $C_1 = \frac{1}{6} = C_2$, $C_3 = C_4 = \frac{1}{3}$

$C_2p_1 = \frac{1}{2} = C_2q_1$, $C_3p_2 = C_3q_2 = \frac{1}{2}$

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)`
   - `t_span`: int. time interval
   - `t_eval`: 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 (Guess) 2) Shoot (Construct path) 3) Compare (Error check & repeat)

This method turn the problem into a initial value problem!

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

3) Compare the given f_b & our obtained f_β and adjust a new guess. Repeat until "convergence" (Compare & iterate)

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

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

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

Example: $\frac{d^2 f(x)}{dx^2} = -10$, $f(0) = 500$, $f(20) = 100$

we need to reduce its order $\Rightarrow \vec{S}(x) = \begin{bmatrix} f(x) \\ f'(x) \end{bmatrix}$, $\frac{d\vec{S}}{dx} = \begin{bmatrix} f'(x) \\ f''(x) \\ -10 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{10}{S_2} & 0 \end{bmatrix} \vec{S}(x)$ →

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 $S_{1,shoot}(x=20) = f_{shoot}(x=20)$ with $f(20)$, change guess & repeat.

Then, we use optimization to find the root of $S_{1,shoot}(x=20) - f(x=20) = 0$

↳ 1) Define obj_fun(v_guess):

```
sol = solve_ivp(slope, [x_00, x_20], np.asarray(S(0), dtype=object),
                method='RK45', teval = 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_00

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:

$$\frac{d^2f}{dx^2} = -10 \quad \Rightarrow \quad f(0) = 500 \qquad f(20) = 100$$

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

↓ in the ODE

Now, we can express this in matrix form as

$$\begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots \\ 0 & 1 & -2 & 1 & \cdots \\ 0 & 0 & 1 & -2 & \cdots \\ \vdots & \cdots & 0 & 0 & 1 \\ 0 & \cdots & \cdots & \cdots & 1 \end{pmatrix} \begin{pmatrix} f_0 = f(0) \\ f_1 \\ \vdots \\ f_{n-1} \\ f_n = f(20) \end{pmatrix} = \begin{pmatrix} 500 \\ -10h^2 \\ -10h^2 \\ \vdots \\ -10h^2 \\ 100 \end{pmatrix}$$

Finally we solve this system with any linear algebra method! $Ax = b$

$\Rightarrow x = A^{-1}b$