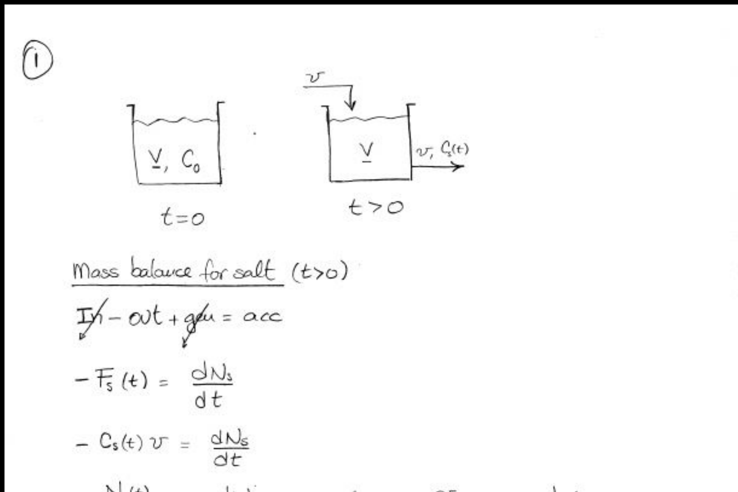
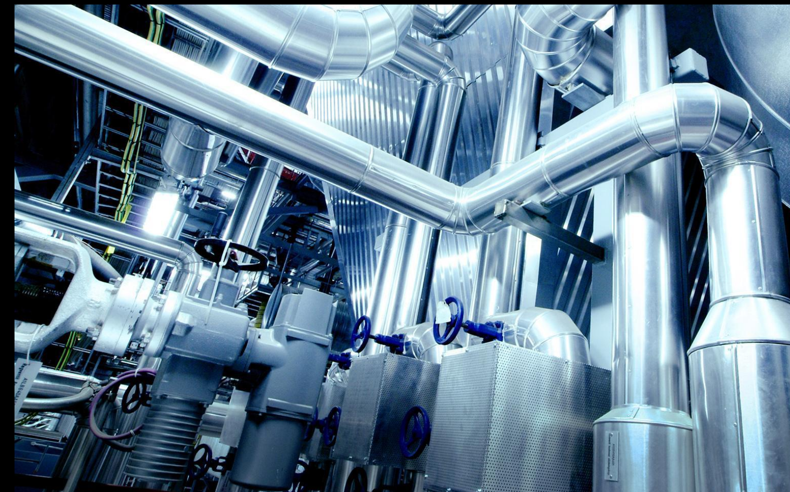




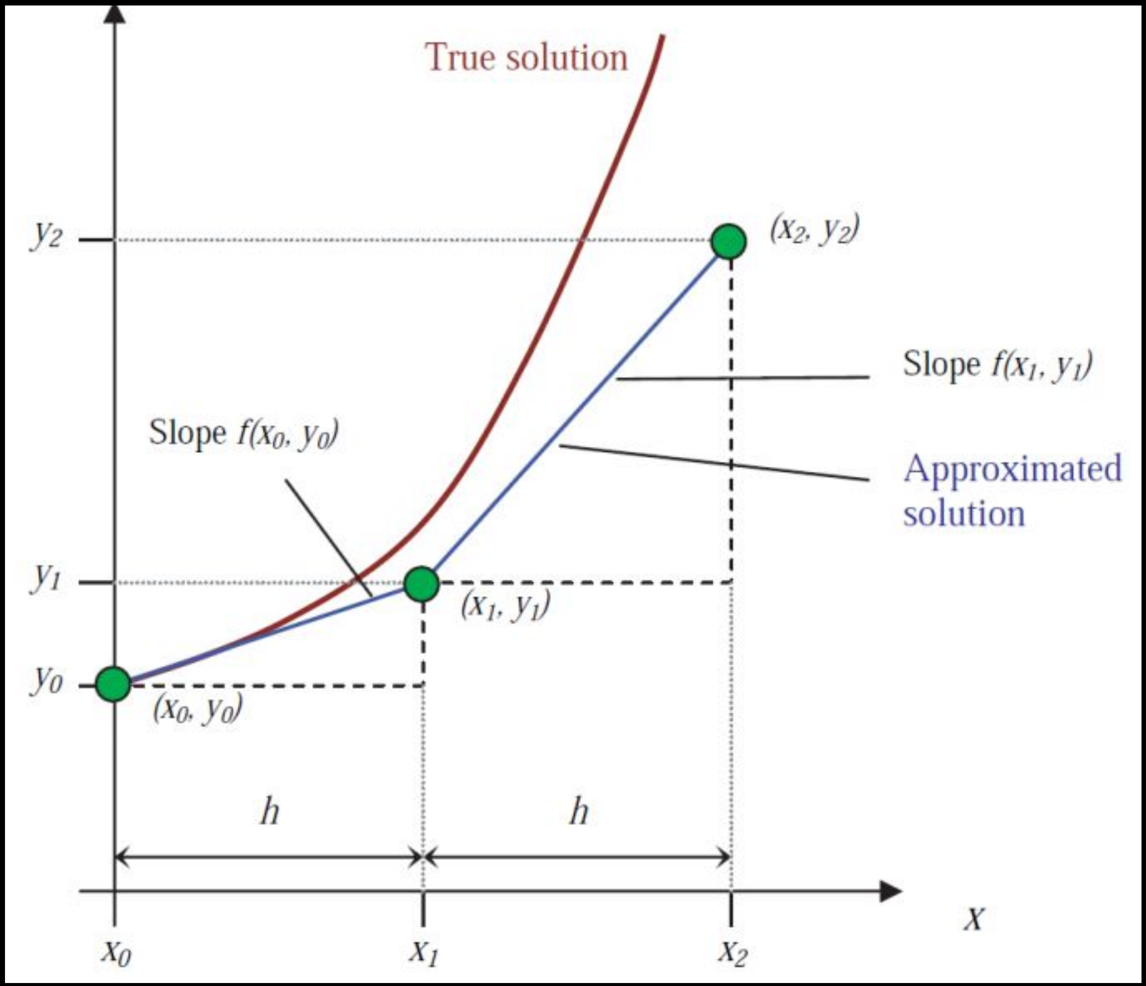
ChE352

Numerical Techniques for Chemical Engineers

Professor Stevenson

Lecture 10

Recall: Initial Value Problems



$$\boxed{WE\ WANT\ y(t)}$$

$$\frac{dy}{dt} = f(t, y)$$

$$a \le t \le b, \quad y(a) = \alpha$$

$$t, y \in \mathbb{R}, \quad y : \mathbb{R} \to \mathbb{R}$$

$$f : \mathbb{R}^2 \to \mathbb{R}$$

$$f\ continuous$$

Why can't we use trapezoidal integration?
What method can we use instead?

Can you find more IVP examples?

- Anything involving <u>rate of change</u>
  - Reaction rates
  - F = ma
  - Epidemics
  - Time-dependent Schrodinger equation

$$i\hbar\frac{\partial}{\partial t}\Psi(x,t) = \left[-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x,t)\right]\Psi(x,t)$$

- **Other examples?**
- We define dy/dt = f(t, y) because f(t, y) is the function we actually <u>have</u> in IVPs
  - y is the function we <u>want</u>

Recall: Euler's Method

$$y\left(t_{i+1}\right) \approx y\left(t_i\right) + hf\left(t_i, y\left(t_i\right)\right)$$

Pronounced the same as "oiler"

Solve the IVP by taking steps along the derivative

Recall: Euler's Method

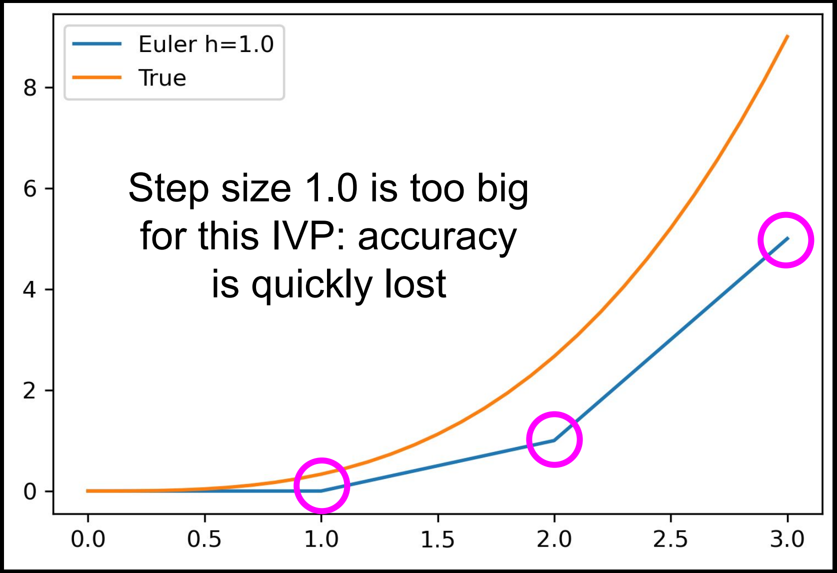
$$y(t_{i+1}) \approx y(t_i) + hf\left(t_i, y(t_i)\right)$$

Example: $f(t) = t^2$ $t_0 = 0$ $y(t_0) = 0$ $h = 1$

$t_1 = h = 1$
$y(1) \approx 0 + 1 * 0^2 = 0$

$t_2 = 2 * h = 2$
$y(2) \approx 0 + 1 * 1^2 = 1$

$t_3 = 3 * h = 3$
$y(3) \approx 1 + 1 * 2^2 = 5$



Euler's Method to get all w[i]

We can define a vector of "time" (call it "t") and calculate our approximate y(t) (aka "w") by iterating forwards in "time" from t_0 = a:

$$t = \begin{bmatrix} t_0 \\ t_1 \\ t_2 \\ \vdots \\ t_{N-1} \end{bmatrix} = \begin{bmatrix} a \\ a+h \\ a+2h \\ \vdots \\ a+(N-1)h \end{bmatrix}, \quad w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_{N-1} \end{bmatrix} = \begin{bmatrix} y(t_0) \\ w_0 + hf(t_0, w_0) \\ w_1 + hf(t_1, w_1) \\ \vdots \\ w_{N-2} + hf(t_{N-2}, w_{N-2}) \end{bmatrix}$$

$t, w \in \mathbb{R}^N$

Why "time" in quotes?

What if we want in-between w values?

Activity: Euler in Python (15 minutes)

$$\boxed{y(t_{i+1}) \approx y(t_i) + hf\left(t_i, y(t_i)\right)}$$ Euler's method

Write a Python function which implements Euler's method for the IVP for this reaction:

$$\frac{dC_{EB}}{d\tau} = -k_f C_{EB}, \quad C_{EB}(\tau = 0) = C_{EB}^o$$

Assume: $k_f = 1.0$, $C^0_{EB} = 2.0$, $\tau_{final} = 10.0$

Use step size h = 0.01. Does the h value matter?

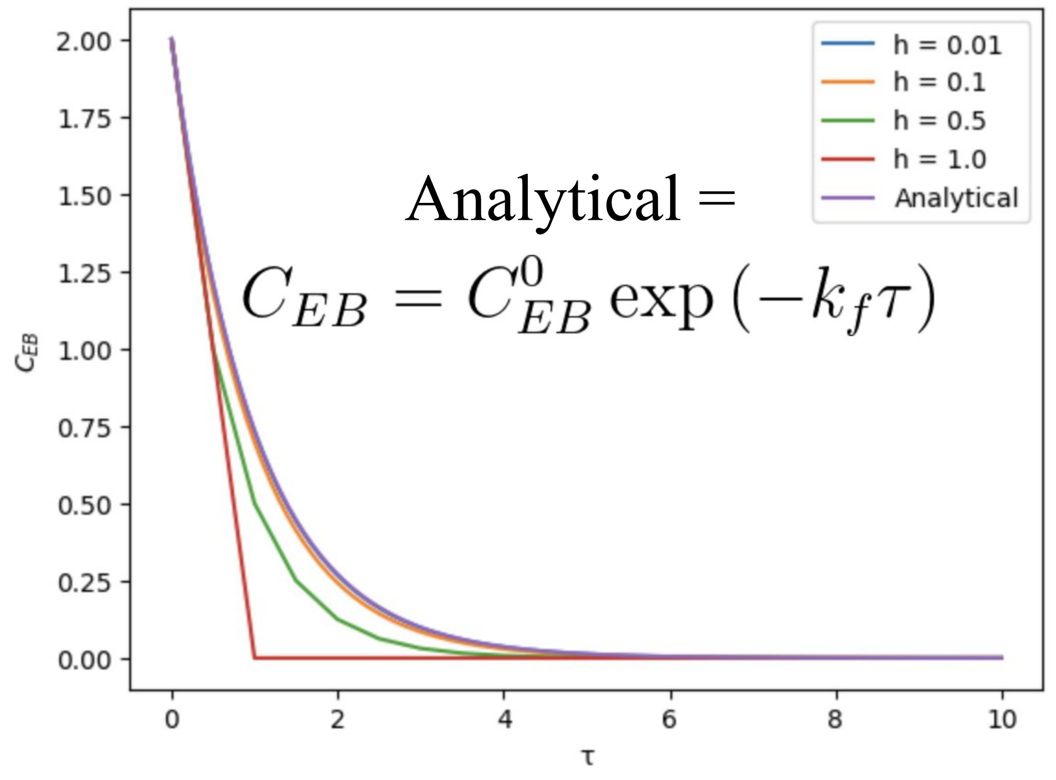
Make a list of your approximate C_{EB} at each step, and if you have time, plot your results vs t

Solution: Euler in Python

$$\boxed{y(t_{i+1}) \approx y(t_i) + hf\left(t_i, y(t_i)\right)}$$ Euler's method

$$\frac{dC_{EB}}{d\tau} = -k_f C_{EB}, \quad C_{EB}(\tau = 0) = C_{EB}^o$$

Euler solution is nearly exact at small dt



Euler solution goes bad fast at large dt

Is there a better IVP method?

- Euler's Method is straightfoward, works if you can afford a small h
    - Local error $O(h^2)$, global error $O(h)$
- But we want better than $O(h)$
- What is *local error* vs *global error*?
- Why is global error 1/h times bigger?
- Why can't we always make h smaller?
- How can we make a better method?
    - Consider where Euler's Method comes from

Taylor Methods of Order n

- Euler's method uses just the linear Taylor terms, but we could use up to any n:

$$y(t) = \boxed{y(t_i) + (t - t_i)\, y'(t_i)} + \boxed{\frac{(t-t_i)^2}{2} y''(t_i) + \ldots + \frac{(t-t_i)^n}{n!} y^{(n)}(t_i)} + \boxed{\frac{(t-t_i)^{n+1}}{(n+1)!} y^{(n+1)}(\xi_i)}$$

$$\Rightarrow \quad y_{i+1} = \boxed{y_i + hf(t_i, y_i)} + \boxed{\frac{h^2}{2} f'(t_i, y_i) + \ldots + \frac{h^n}{n!} f^{(n-1)}(t_i, y_i)} + \boxed{\frac{h^{n+1}}{(n+1)!} f^{(n)}(\xi_i, y(\xi_i))}$$

Linear terms — Quadratic and higher terms — Error term

- By definition, $y'(t) = f(t, y)$ ⟵ We always have this in an IVP
- 2nd derivative: $y''(t) = f'(t, y)$ ⟵ Might not have this
- n-th derivative: $y^n(t) = f^{n-1}(t, y)$ ⟵ Good luck

Taylor Methods of Order n

- Euler's method uses just the linear Taylor terms, but we could use up to any n:

$$y(t) = \boxed{y(t_i) + (t - t_i)\,y'(t_i)} + \boxed{\frac{(t-t_i)^2}{2}\,y''(t_i) + \ldots + \frac{(t-t_i)^n}{n!}\,y^{(n)}(t_i)} + \boxed{\frac{(t-t_i)^{n+1}}{(n+1)!}\,y^{(n+1)}(\xi_i)}$$

$$\Rightarrow \quad y_{i+1} = \boxed{y_i + hf(t_i, y_i)} + \boxed{\frac{h^2}{2} f'(t_i, y_i) + \ldots + \frac{h^n}{n!} f^{(n-1)}(t_i, y_i)} + \boxed{\frac{h^{n+1}}{(n+1)!} f^{(n)}(\xi_i, y(\xi_i))}$$

$$y_{i+1} \approx \boxed{y_i + hf(t_i, y_i)} + \boxed{\frac{h^2}{2} f'(t_i, y_i) + \ldots + \frac{h^n}{n!} f^{(n-1)}(t_i, y_i)}$$

- If we use a series of order n, the <u>local error</u> for each step is $O(h^{n+1})$ (Why?)
- <u>Global error</u> after all steps is $O(h^n)$ (Why?)

Activity: 2nd Order Taylor Methods

$$\boxed{y_{i+1} \approx y_i + hf\left(t_i, y_i\right) + \frac{h^2}{2} f'\left(t_i, y_i\right) + \ldots + \frac{h^n}{n!} f^{(n-1)}\left(t_i, y_i\right)}$$

Translate the Taylor polynomial formula above into an iterative step for the 2nd-order Taylor method for IVPs, giving w_{i+1} in terms of w_i, t_i, f, f', and h.

Use your general expression to define the iterative step w_{i+1} for this IVP:

$$y' = y - t \qquad t_0 = 0 \qquad y(0) = e + 1$$

Leave your expression in terms of h (Why?)

Answer: 2nd Order Taylor Methods

y' = y – t $t_0 = 0$ y(0) = e + 1

Euler's method:

$$w_0 = e + 1$$

$$w_{i+1} = w_i + h(w_i - t_i) = (h+1)w_i - ht_i \quad (i = 0 \ldots N-2)$$

2nd order Taylor:

$$w_{i+1} = w_i + hf(t_i, w_i) + \frac{h^2}{2} f'(t_i, w_i) = w_i + h(w_i - t_i) + \frac{h^2}{2}\frac{d}{dt}(w - t)_i$$

$$= w_i + hw_i - ht_i + \frac{h^2}{2}(w_i - t_i) - \frac{h^2}{2} = \boxed{\left(\frac{h^2}{2} + h + 1\right)w_i - h\left(\frac{h}{2} + 1\right)t_i - \frac{h^2}{2}}$$

What are some drawbacks of this method?

The problem with f'

- Taylor methods gain more accuracy by using more derivatives of f
  - Recall: $y^n(t) = f^{n-1}(t, y)$
- But derivatives of f are rarely available
- Can we approximate $f'(t, y)$ using the values of $f(t, y)$? How?
- The resulting methods are the most popular IVP solvers: **Runge-Kutta**

Runge-Kutta Methods: RK2

Use 2D Taylor series & the chain rule to **find** $f'(t_i, y_i)$, with Δt = h/2 and $\Delta y = \Delta t\, f(t_i, y_i)$. Then plug $f'(t_i, y_i)$ into the 2nd order Taylor method.

$$f\left(t+\Delta t, y+\Delta y\right) \approx f\left(t, y\right) + \left[\Delta t\left(\frac{\partial f}{\partial t}\right)_{t,y} + \Delta y\left(\frac{\partial f}{\partial y}\right)_{t,y}\right]$$

2D Taylor series in y, t

Chain rule gives $f'(t_i, y_i)$

$$\boxed{f'\left(t_i, y_i\right)} = \left(\frac{\partial f}{\partial t}\right)_{t_i,y_i} + \left(\frac{\partial f}{\partial y}\right)_{t_i,y_i}\left(\frac{dy}{dt}\right)_{t_i}$$

$$y_{i+1} \approx y_i + hf\left(t_i, y_i\right) + \frac{h^2}{2}\boxed{f'\left(t_i, y_i\right)}$$

2nd order Taylor method needs $f'(t_i, y_i)$

Runge-Kutta Methods: RK2

$$f\left(t_{i+1}, y_{i+1}\right) \approx f\left(t_i, y_i\right) + \left[\Delta t \left(\frac{\partial f}{\partial t}\right)_{t_i, y_i} + \Delta y \left(\frac{\partial f}{\partial y}\right)_{t_i, y_i}\right]$$

2D Taylor series in y, t

$$= f\left(t_i, y_i\right) + \boxed{\frac{h}{2}} \left(\frac{\partial f}{\partial t}\right)_{t_i, y_i} + \boxed{\frac{h}{2} f\left(t_i, y_i\right)} \left(\frac{\partial f}{\partial y}\right)_{t_i, y_i}$$

Same as chain rule!

$$= f\left(t_i, y_i\right) + \frac{h}{2}\left[\left(\frac{\partial f}{\partial t}\right)_{t_i, y_i} + \boxed{f\left(t_i, y_i\right)} \left(\frac{\partial f}{\partial y}\right)_{t_i, y_i}\right] = f\left(t_i, y_i\right) + \frac{h}{2}\underbrace{\left[\left(\frac{\partial f}{\partial t}\right)_{t_i, y_i} + \boxed{\left(\frac{dy}{dt}\right)_{t_i}} \left(\frac{\partial f}{\partial y}\right)_{t_i, y_i}\right]}_{\text{Same as chain rule!}}$$

By definition: $f'(t, y) = dy/dt$

Chain rule gives $f'(t_i, y_i)$ →

$$\boxed{f'\left(t_i, y_i\right)} = \left(\frac{\partial f}{\partial t}\right)_{t_i, y_i} + \left(\frac{\partial f}{\partial y}\right)_{t_i, y_i} \left(\frac{dy}{dt}\right)_{t_i}$$

$$f'\left(t_i, y_i\right) \approx \frac{2}{h}\left[f\left(t_{i+1}, y_{i+1}\right) - f\left(t_i, y_i\right)\right]$$

Runge-Kutta Methods: RK2

$$\boxed{f'\left(t_i, y_i\right)} \approx \frac{2}{h}\left[f\left(t_{i+1}, y_{i+1}\right) - f\left(t_i, y_i\right)\right],$$

Given f', we can plug it into the 2nd order Taylor IVP method

$$w_{i+1} = w_i + hf\left(t_i, w_i\right) + \frac{h^2}{2}\boxed{f'\left(t_i, w_i\right)} \rightarrow$$

$$w_{i+1} = w_i + hf\left(t_i, w_i\right) + \frac{h^2}{2}\left(\frac{2}{h}\right)\left[f\left(t_{i+1}, w_{i+1}\right) - f\left(t_i, w_i\right)\right]$$

$$= w_i + hf\left(t_i, w_i\right) + h\left[f\left(t_{i+1}, w_{i+1}\right) - f\left(t_i, w_i\right)\right] \rightarrow$$

$$w_{i+1} = w_i + hf\left(t_i, w_i\right) + hf\left(t_{i+1}, w_{i+1}\right) - hf\left(t_i, w_i\right)$$

$$= w_i + hf\left(t_{i+1}, w_{i+1}\right) = w_i + hf\left(t_i + \frac{h}{2}, w_i + \frac{h}{2} f\left(t_i, w_i\right)\right) \rightarrow$$

$$\boxed{w_{i+1} = w_i + hf\left(t_i + \frac{h}{2}, w_i + \frac{h}{2} f\left(t_i, w_i\right)\right)}$$

RK2, aka "midpoint method for IVPs"

Activity: RK2 in Python (10 minutes)

$$\boxed{y\left(t_{i+1}\right) \approx y\left(t_i\right) + hf\left(t_i, y\left(t_i\right)\right)}$$ Euler's method

Copy your Python IVP solver from before and change it to RK2:

$$\boxed{w_{i+1} = w_i + hf\left(t_i + \frac{h}{2}, w_i + \frac{h}{2} f\left(t_i, w_i\right)\right)}$$ RK2

Make a list of your approximate C_{EB} at each step, and if you have time, plot your results vs t

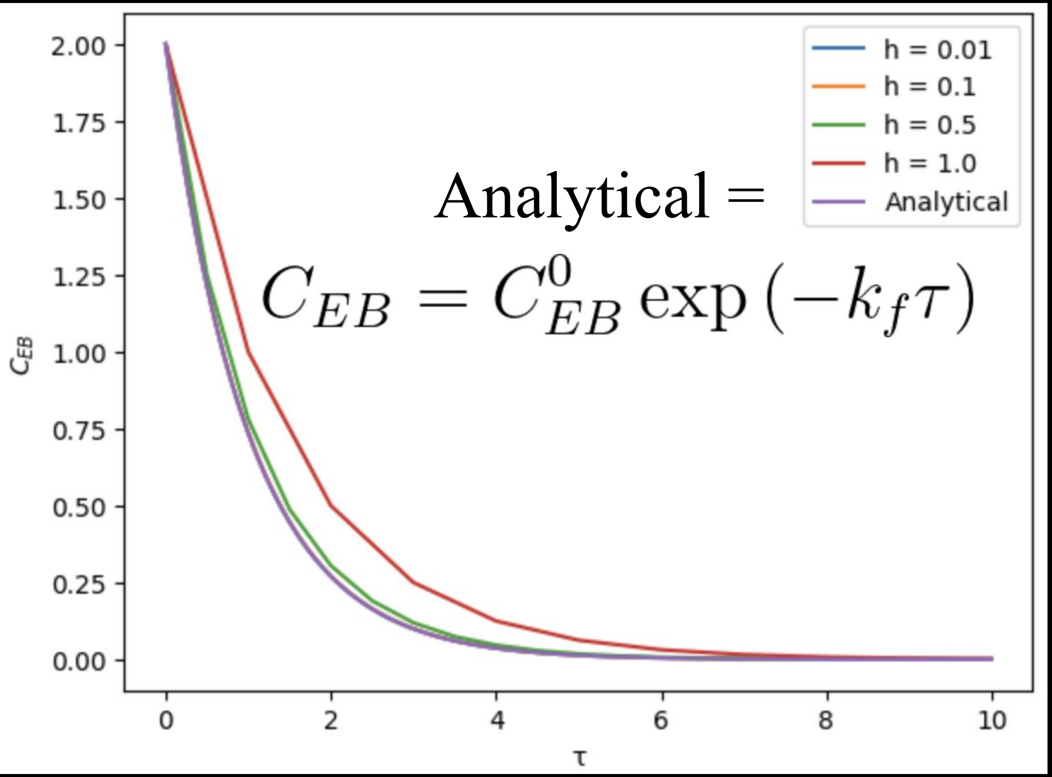
How does the dependence on h change?

Solution: RK2 in Python

$$\boxed{w_{i+1} = w_i + hf\left(t_i + \frac{h}{2}, w_i + \frac{h}{2} f\left(t_i, w_i\right)\right)} \quad \text{RK2}$$

$$\frac{dC_{EB}}{d\tau} = -k_f C_{EB}, \quad C_{EB}\left(\tau = 0\right) = C_{EB}^{o}$$

RK2 solution is nearly exact at small dt



RK2 solution does not go bad so fast

Better Runge-Kutta?

- Different values for Δt and Δy in 2D Taylor make new IVP methods (F&B 185-187)
- Order 2 methods have global approximation error of $O(h^2)$
- Most common RK method for solving IVPs is order 4, which uses the Taylor terms up to h^4
- This method is called <u>RK4</u> or just <u>The Runge-Kutta Method</u> for IVPs
- Given this description, what is the big-O of local & global error for RK4?

"The" Runge-Kutta Method: RK4

$$w_{i+1} = w_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

$$Where: \quad k_1 = hf\left(t_i, w_i\right)$$

$$k_2 = hf\left(t_i + \frac{h}{2}, w_i + \frac{1}{2}k_1\right)$$

$$k_3 = hf\left(t_i + \frac{h}{2}, w_i + \frac{1}{2}k_2\right)$$

$$k_4 = hf\left(t_{i+1}, w_i + k_3\right)$$

- Like RK2 but more
- Global error $O(h^4)$
- Requires 4 calls to f (t, y) per step
- Don't need f'(t, y)
- Usually the sweet spot for accuracy

Why stop at RK4?

- The main cost for using an IVP algorithm is the calls to function f – fewer is better
- Euler needs 1 function evaluation per step
- RK4 needs 4
- RK4 is only useful if it allows step sizes over 4x bigger, with the same accuracy (**it does**)
- Table on p. 188 of F&B shows that <u>RK4 is superior to lower *and* higher order methods</u> by this metric under reasonable assumptions

Activity: Local Error in RK4

1. Use RK4 to estimate y(0.1) for this IVP:

$y' = y - t \qquad t_0 = 0 \qquad y(0) = e + 1 \qquad h = 0.1$

2. Just as a demonstration of the error, compare your approximation to the exact answer $y(t) = e^{t+1} + t + 1$ to get the actual local relative approximation error. Is it similar in scale to h^5?

Answer: Local Error in RK4

$$w_0 = e + 1, \quad h = 0.1$$

$$w_1 = e + 1 + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) = \boxed{4.104165794}$$

$$\mathit{Where:} \quad k_1 = 0.1\left[w_0 - t_0\right] = 0.1e + 0.1$$

$$k_2 = (0.1)\left[w_0 + \frac{1}{2}k_1 - t_0 - \frac{h}{2}\right] = 0.105e + 0.1$$

$$k_3 = (0.1)\left[w_0 + \frac{1}{2}k_2 - t_0 - \frac{h}{2}\right] = 0.10525e + 0.1$$

$$k_4 = (0.1)\left[w_0 + k_3 - t_1\right] = 0.110525e + 0.1$$

$$y_1 = e^{1.1} + 1.1 \quad \rightarrow \quad \boxed{error = 5.61 \times 10^{-8}} \quad (really\ tiny)$$

SciPy generic IVP solver: solve_ivp

```
from scipy.integrate import solve_ivp
sol = solve_ivp(fun, (t0, t_end), [y0])
plt.plot(sol.t, sol.y[0], label='RK45')
```

- Uses RK4 but with dynamic h, with an error estimate based on RK5 - known as RK4(5)
  - Also has other, specialized methods
- Can solve for multi-dimensional y in f(t, y)
- Returns an object containing data about the solution, including sol.t, sol.y, & sol.success

IVP Systems

- 1D problems are common, but so are IVPs with multiple outputs:

$$\frac{dN_S}{dz} = \frac{k_f N_{EB}}{v} - \frac{k_r N_S N_H}{v} = R_S^{'}, \quad N_S(z=0) = vC_S^o$$

$$\frac{dN_{EB}}{dz} = -R_S^{'}, \quad N_{EB}(z=0) = vC_{EB}^o$$

$$\frac{dP}{dz} = -\frac{\rho v^2}{d_p}\left(\frac{1-\varepsilon}{\varepsilon^3}\right)\left[\frac{150(1-\varepsilon)}{\mathrm{Re}_p} + 1.75\right], \quad \frac{dN_W}{dz} = 0, \quad N_S = N_H$$

$$P\hat{V} = ZRT \quad \Rightarrow \quad \rho = \frac{PM}{ZRT} \quad \Rightarrow \quad v = \frac{ZRT}{P}\left(N_{EB} + N_S + N_H + N_W\right)$$

Where are the dependent variables here?

- We need output to be a <u>vector</u> instead of a scalar - u now instead of y

Numerical Soln. of IVP Systems

Suppose your problem now looks like this:

$$t_0 \le t \le t_{\max}$$

$$\frac{du_1}{dt} = f_1\left(t, u_1, u_2, \ldots, u_m\right), \quad u_1\left(t = t_0\right) = a_1$$

$$\frac{du_2}{dt} = f_2\left(t, u_1, u_2, \ldots, u_m\right), \quad u_2\left(t = t_0\right) = a_2 \quad \Rightarrow$$

$$\vdots$$

$$\frac{du_m}{dt} = f_m\left(t, u_1, u_2, \ldots, u_m\right), \quad u_m\left(t = t_0\right) = a_m$$

a, not α

$$\boxed{\begin{gathered} \frac{du(t)}{dt} = f\left(t, u(t)\right), \\ u\left(t = t_0\right) = a, \\ t_0 \le t \le t_{\max} \\ u : \mathbb{R} \to \mathbb{R}^m, \\ f : \mathbb{R}^{m+1} \to \mathbb{R}^m, \\ t \in \mathbb{R}, \quad a \in \mathbb{R}^m \end{gathered}}$$

Vector function

Vector function

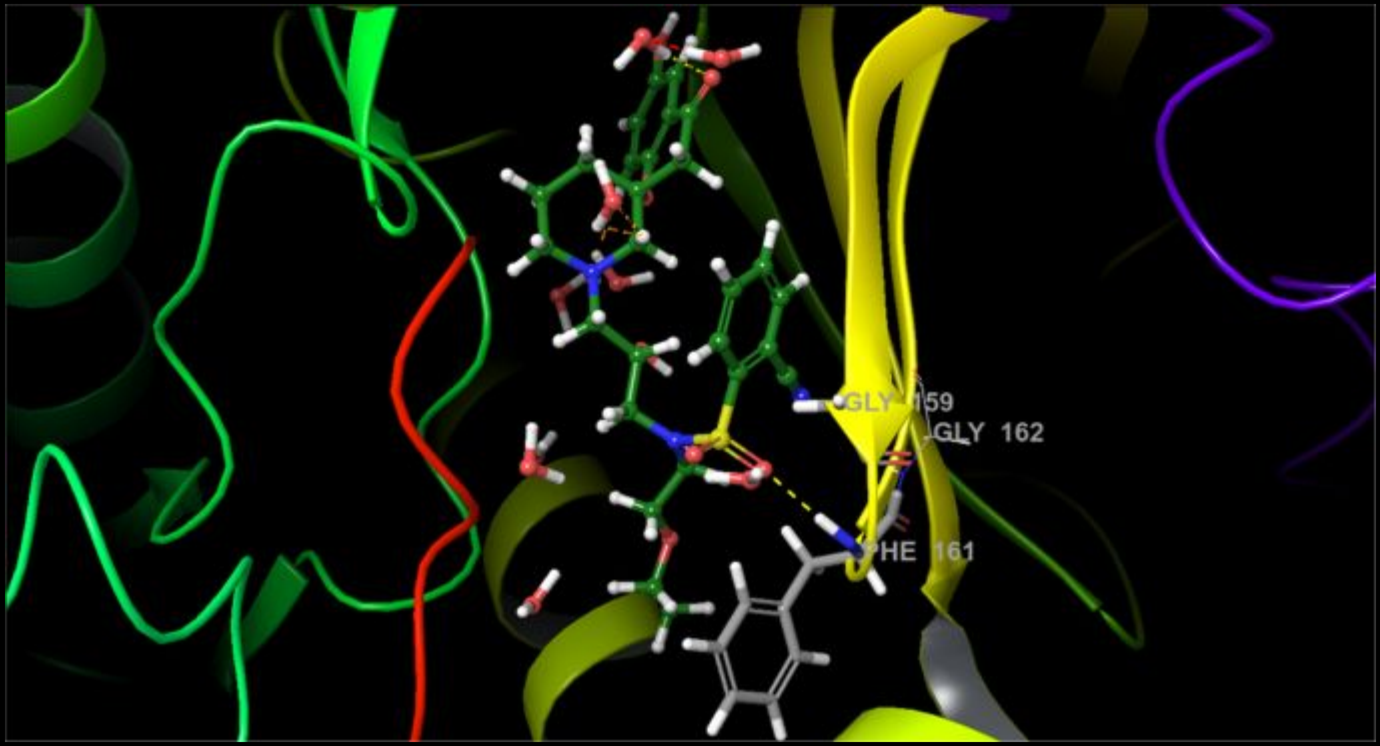
Same methods work!

IVP Systems in Python

```
from scipy.integrate import solve_ivp
def fun(t, u):  # 3-D IVP
    C_A, C_B, C_C = u
    ... calculate du/dt here ...
    return dAdt, dBdt, dCdt
sol = solve_ivp(fun, (t0, t_final), u0)
plt.plot(sol.t, sol.y[0], label='[A]')
plt.plot(sol.t, sol.y[1], label='[B]')
plt.plot(sol.t, sol.y[2], label='[C]')
```

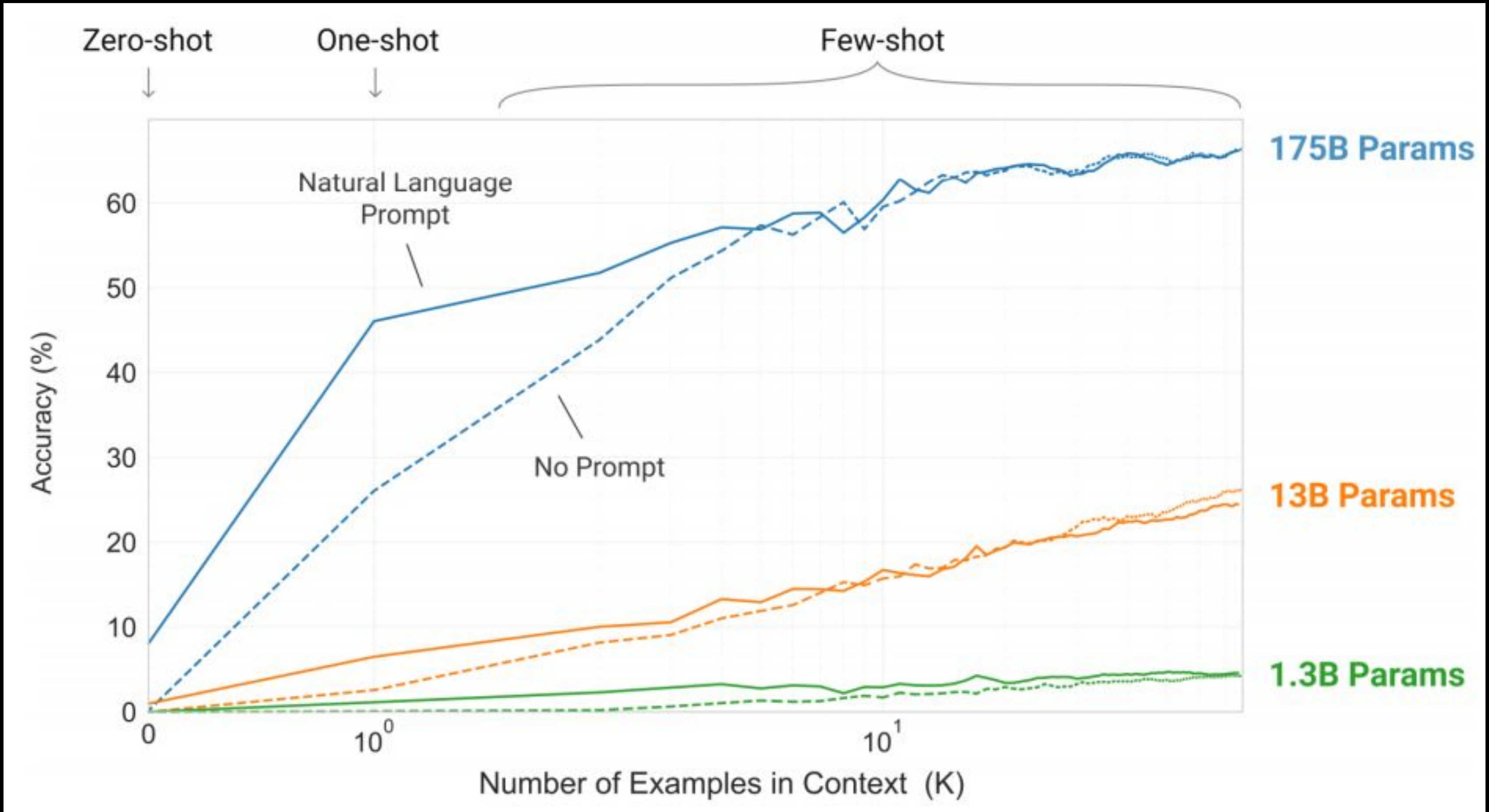
Million+ Dimension IVP Systems

- IVPs often scale to millions of dimensions
- Example: *molecular dynamics*, every [x, y, z] of every atom is another dimension of w(t)
- Same techniques apply, just more compute



10^{12}+ Dimension IVP Systems

- Machine learning all known text / images
- *Same techniques apply, just more compute*



Activity: Coding RK4

- Write a function that calculates the next step of RK4:

```
def rk4(f, ti, wi, h):
    ...your code...
    return w_next
```

- Try it with this IVP:

```
def fun(t, w):
    return w - t
t0 = 0; y0 = np.e+1
```

$$w_0 = y(a) = \alpha$$

$$w_{i+1} = w_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

$$Where: \quad k_1 = hf\left(t_i, w_i\right)$$

$$k_2 = hf\left(t_i + \frac{h}{2}, w_i + \frac{1}{2}k_1\right)$$

$$k_3 = hf\left(t_i + \frac{h}{2}, w_i + \frac{1}{2}k_2\right)$$

$$k_4 = hf\left(t_{i+1}, w_i + k_3\right)$$

When you've got it, compare vs scipy.integrate.solve_ivp

Pre-reading for next week

Predictor-corrector & adaptive methods for IVPs, higher-order IVPs, stiff IVPs:
PNM 22.6-7, F&B 5.6-8.

Verlet integration:
https://www.algorithm-archive.org/contents/verlet_integration/verlet_integration.html