Unit 3:

Numerical ODEs

Most Ordinary Differential Equations (ODEs) are not solvable on paper. Here's how we deal with them using computers.

Let's start with an example.

Midterm

Example: Population Modeling

Let $p(t)$ be the population at time t.

$$p(2000) = 30,700,000$$

We will refer to $p(t)$ as a "state variable". What affects population?

birth rate $b = 0.0109 \frac{\text{people}}{\text{people} \cdot \text{year}}$

immigration $I = 300,000 \frac{\text{people}}{\text{year}}$

death rate $d = 0.0075 \frac{\text{people}}{\text{people} \cdot \text{year}}$

emigration $m = 0.002 \frac{\text{people}}{\text{people} \cdot \text{year}}$

Change in $p(t)$ in 1 year $=$ births $-$ deaths $+$ immigration $-$ emigration

$$= b\,p(t) - d\,p(t) + I - m\,p(t)$$

$$= \underbrace{(b-d-m)}_{r}\,p(t) + I$$

$$\frac{dp(t)}{dt} = r\,p(t) + I \quad (1)$$

This is a differential equation (DE) because it involves a derivative. If we could find a function $p(t)$ that satisfies (1), then $p(t)$ would be called the "solution" of the DE.

$p(t_0) = p_0$

$p(t) = \left(p_0 + \frac{I}{r}\right) e^{r(t-t_0)} - \frac{I}{r}$ is a solution of (1)

called the "solution" of the DE.

$p(t_0) = p_0$

eg. $p(t) = \left(p_0 + \frac{I}{r}\right) e^{r(t-t_0)} - \frac{I}{r}$ is a solution of (1)

$LHS = \frac{dp(t)}{dt} = \left(p_0 + \frac{I}{r}\right) r e^{r(t-t_0)}$

$RHS = r\left[\left(p_0 + \frac{I}{r}\right) e^{r(t-t_0)} - \frac{I}{r}\right] + I$

$= r\left(p_0 + \frac{I}{r}\right) e^{r(t-t_0)}$ ✓

Another example:

$y'(x) = y$

$y(0) = 1$

$x \in [0, 2]$

$y'(x) = f(x, y) = y$

solution is $y(x) = ce^x$

$y(0) = 1 = ce^0 = c$

$\therefore y(x) = e^x$

How about

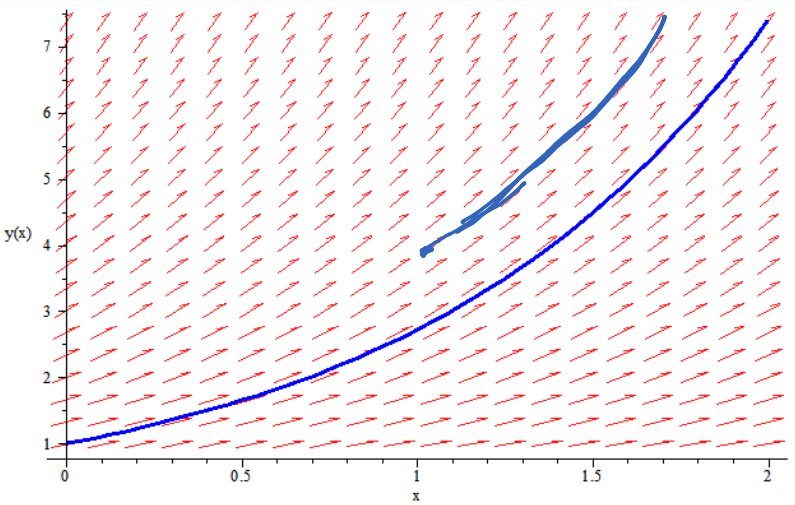
$y'(x) = y$

$y(1) = 4$

$y(1) = 4 = ce^1$

$\therefore c = \frac{4}{e}$

$\therefore y(x) = 4e^{x-1}$



Initial Value Problems (IVP)

An Initial Value Problem consists of two parts:

1) Dynamics equations, and
2) An initial state.

Standard Form for First-Order IVPs

Data for dynamics

m = # of state variables

M = $m \times m$ matrix

$f(t,z)$ = m-vector-valued function of $t, z_1, \ldots, z_m$ (time, and m state values)

Data for initial state

t_0 = start time

$z^{(i)}$ = initial state (m-vector)

The IVP is

$$M \frac{dz(t)}{dt} = f(t,z)$$

$$z(t_0) = z^{(i)}$$

Example: Population Model

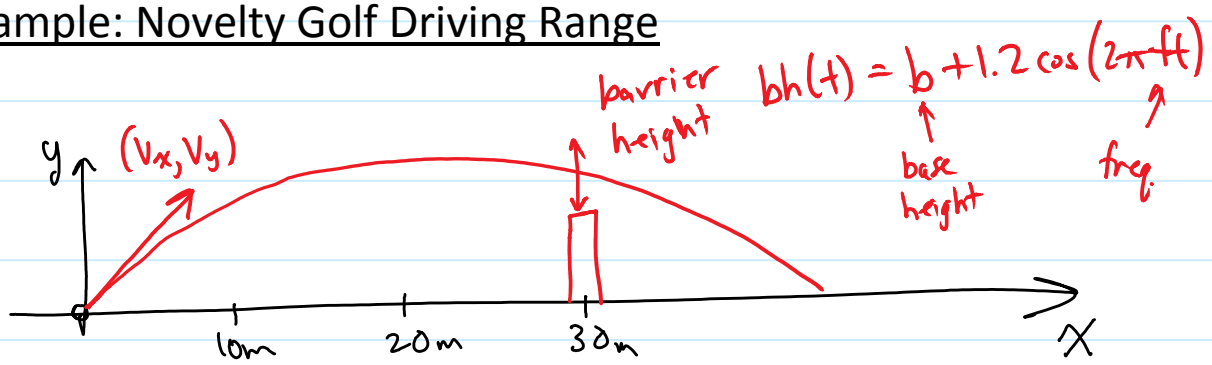
$m = 1$

$M = I = 1$

(1) $f(t,z) = rz + I$ ... $\frac{dp(t)}{dt} = rp(t) + I$

(2) $z(2000) = 30,700,000$

Example: Novelty Golf Driving Range



Ball starts at $(x(0), y(0)) = (0, 0)$
Golfer hits the ball with initial velocity vector (V_x, V_y).

Dynamics Model

$$\frac{dx(t)}{dt} = V_x$$

$$\frac{d^2y(t)}{dt^2} = -g \quad \left(g = 9.81 \frac{m}{s^2}\right)$$

This is a 2nd-order differential equation (DE). Standard 1st-order form involves only 1st-order DEs.

Let $z_1 = x$, $z_2 = y$, $z_3 = \frac{dy}{dt}$ $\leftarrow$ $z_3 = \frac{dz_2}{dt}$

$$\frac{dz_1}{dt} = V_x$$

$$\frac{dz_2}{dt} = z_3 \Rightarrow \frac{d}{dt}\begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = f(t,z) = \begin{bmatrix} V_x \\ z_3 \\ -g \end{bmatrix} \quad \& \quad z(0) = \begin{bmatrix} 0 \\ 0 \\ V_y \end{bmatrix}$$

$$\frac{dz_3}{dt} = -g$$

Another example of a higher-order ODE converted into a system of first-order ODEs.

eg. $y'' + 3y' + 4y = \cos x$

Let $z_1 = y$

$z_2 = y' \Rightarrow z_2 = \frac{dz_1}{dt}$

Let $z_1 = y$

$z_2 = y' \Rightarrow \boxed{z_2 = \frac{dz_1}{dt}}$

Then the system becomes

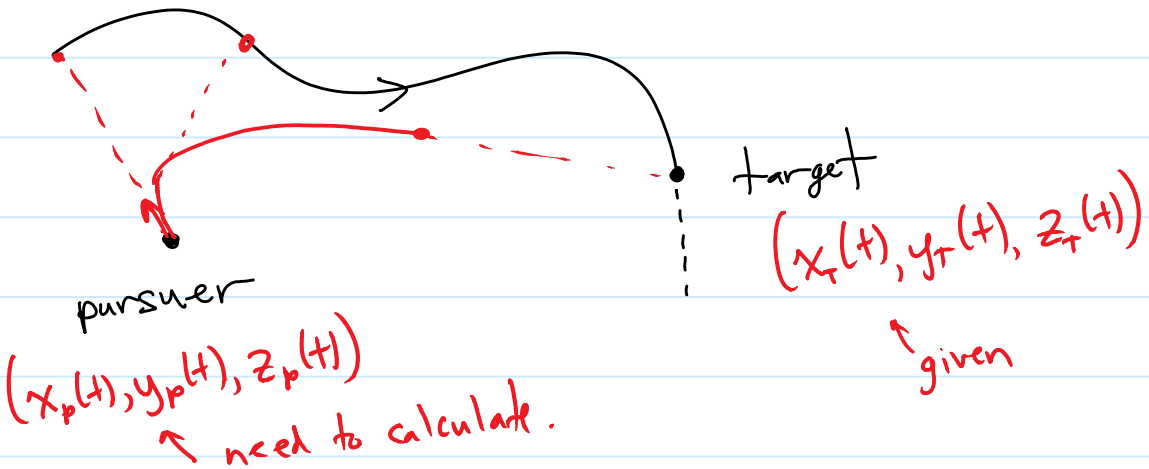
$$\begin{cases} z_1' = z_2 \\ z_2' = \cos x - 3z_2 - 4z_1 \end{cases}$$

Thus

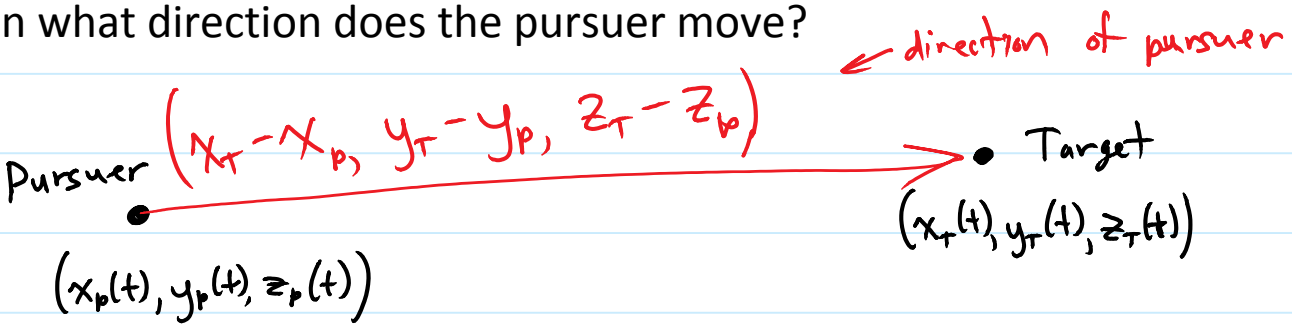
$$\frac{d}{dt}\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \underbrace{\begin{bmatrix} z_2 \\ \cos x - 3z_2 - 4z_1 \end{bmatrix}}_{f(t,z)}$$

Example: Pursuit

Think of a fox chasing a rabbit, or a heat-seaking missile fired at a jet.



The pursuer moves with speed S_p directly at the target at all times. In what direction does the pursuer move?



As a unit vector...

$$\frac{(x_T - x_p,\ y_T - y_p,\ z_T - z_p)}{\sqrt{(x_T - x_p)^2 + (y_T - y_p)^2 + (z_T - z_p)^2}} \leftarrow \text{dist}(t, x_p, y_p, z_p)$$

We multiply it by S_p to get the velocity vector for the pursuer.

$$\therefore \quad \frac{dx_p(t)}{dt} = \frac{S_p}{\text{dist}(t, x_p, y_p, z_p)}(x_T - x_p)$$

$$\frac{dy_p(t)}{dt} = \frac{S_p}{\text{dist}(t, x_p, y_p, z_p)}(y_T - y_p)$$

$$\frac{dz_p(t)}{dt} = \frac{S_p}{\text{dist}(t, x_p, y_p, z_p)}(z_T - z_p)$$

In standard form, $m = 3 \quad M = I$

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} \rightarrow \begin{bmatrix} x_p \\ y_p \\ z_p \end{bmatrix} \quad f(t,z) = \frac{S_p}{\text{dist}(t,z)} \begin{bmatrix} x_T - z_1 \\ y_T - z_2 \\ z_T - z_3 \end{bmatrix} \quad z(0) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Terminal Events

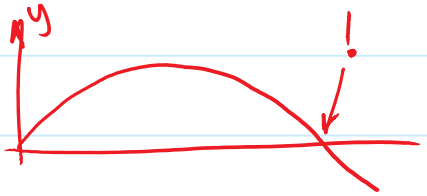
A terminal event occurs when something happens in the simulation that is not incorporated in the dynamics model. For example, a falling object hits the ground, or a prey species goes extinct.

Eg. **Golf Driving Range Model**

We want to terminate the computation when

a. the ball hits the ground

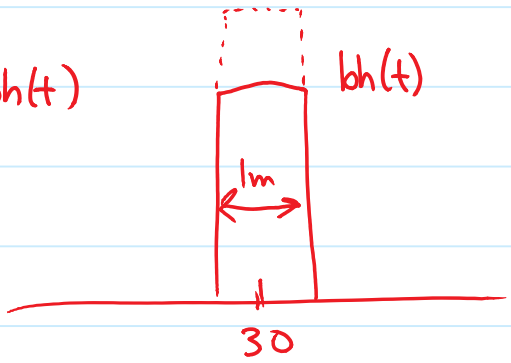
$$E_1(t,z) = z_2 \quad (\text{recall } z_2 = y)$$



b. the ball hits the barrier

b. the ball hits the barrier

$$E_2(t,z) = \begin{cases} -1 & \text{if } |z_1 - 30| \le \frac{1}{2} \ \& \ z \le bh(t) \\ 1 & \text{otherwise} \end{cases}$$



Better (continuous):

$$E_2(t,z) = \max\left(|z_1 - 30| - \tfrac{1}{2},\ z_2 - bh(t)\right)$$

Returns negative iff ball is inside barrier.

Eg. **Pursuit**

Terminate when **the pursuer catches the target.**

$$E(t,z) = dist(t,z) - d_{min}$$

i.e. Pursuer is within d_{min} of the target.

END.

Numerical Solution of ODEs

Almost all DEs are not solvable analytically. That is, rarely do we have a mathematical formula for the solution of a DE. In practice, solutions to DEs are usually approximated numerically.

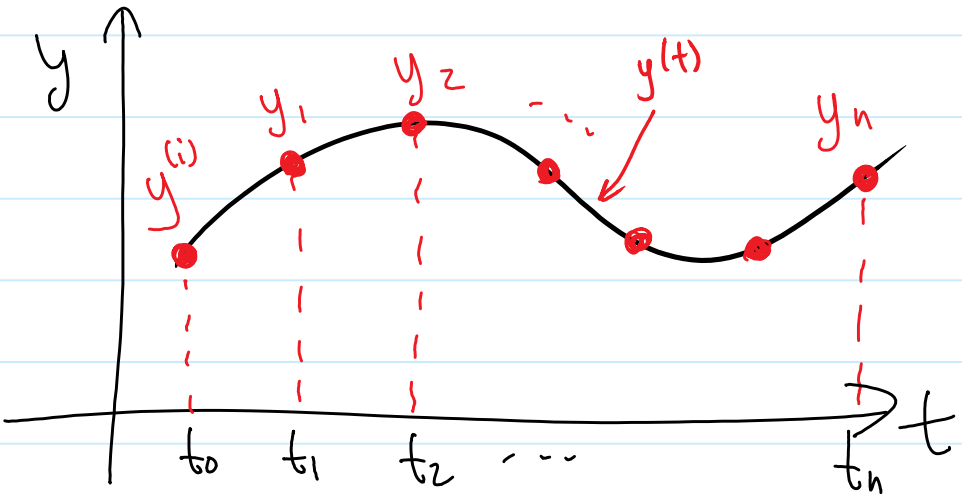
Consider the IVP

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

2) $y(t_0) = y^{(i)}$

If $y(t)$ is the (hypothetical) true solution, we want to find a set of points $\{t_n, y_n\}$ such that

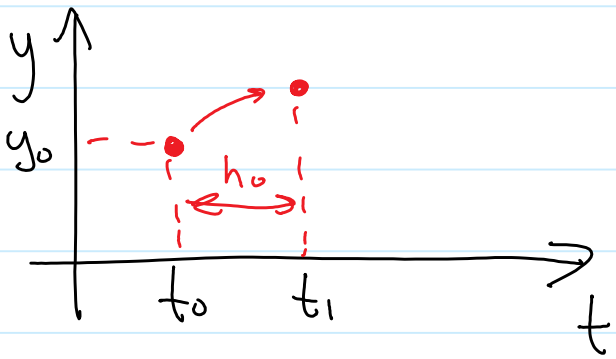
$y_n \approx y(t_n)$



Euler's Method

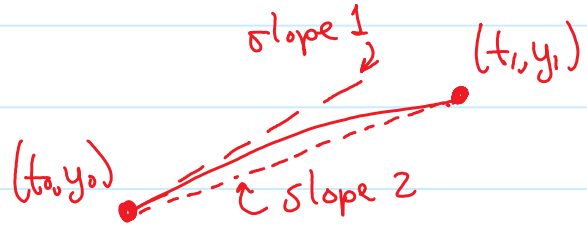
Starting with the initial state, we can approximate the solution by **taking small steps in time (the independent variable).**

Let h_0 be the first time step, so that $t_1 - t_0 = h_0$.



The slope at (t_0, y_0) can be approximated by

$$\underbrace{\frac{y_1 - y_0}{h_0 = t_1 - t_0}}_{\text{(slope 2)}} \cong \underbrace{f(t_0, y_0)}_{\text{(slope 1)}}$$

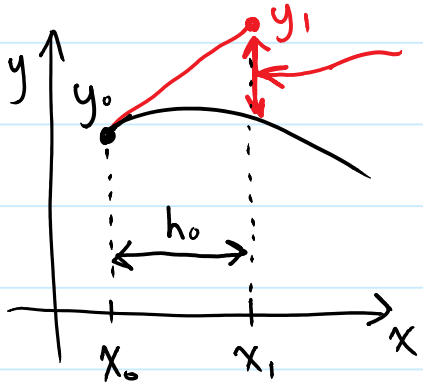


$$y_1 \cong y_0 + h_0 f(t_0, y_0)$$

And in general: $y_{k+1} = y_k + h_k f(t_k, y_k)$

This is called **Euler's Method** (Euler_demo.m)

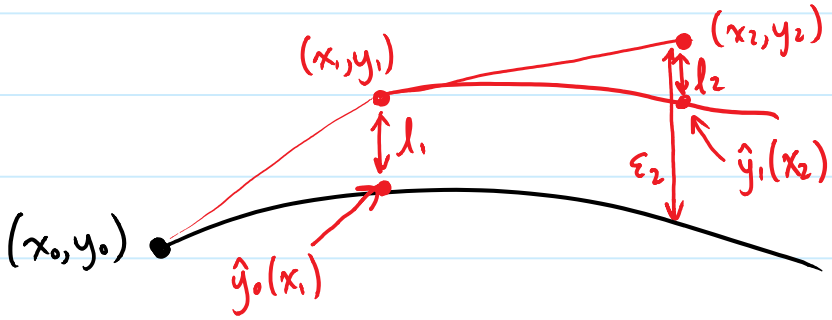
pronounced like "oilers"



The difference between the true solution & the approx. in a single step is called the "local error", denoted l_1.

Local and Global Error

Let us denote the true solution curve through (x_n, y_n) as $\hat{y}_n(x)$.



Local Error: $l_{n+1} = |\hat{y}_n(x_{n+1}) - y_{n+1}|$

Global Error: $\varepsilon_{n+1} = |\hat{y}_0(x_{n+1}) - y_{n+1}|$

↑ solution curve through (x_0, y_0) i.e. $y(x_{n+1})$

The local error of Euler's Method is $O(h^2)$. (euler_error_demo.m)

To integrate a solution through a fixed domain of length c (eg. tspan=[0 2], so $c=2$), the number of steps of length h would be

$N = \frac{c}{h}$ Inversely proportional to h.

For this reason, the global error for Euler's Method is $O(h)$.

Example: Golf Driving Range

$$f(t,z) = \begin{bmatrix} V_x \\ z_3 \\ -g \end{bmatrix} \qquad z(0) = \begin{bmatrix} 0 \\ 0 \\ V_y \end{bmatrix}$$

Let $V_x = 30$, $V_y = 12$, $g = 9.81$

Euler step of size $h_0 = 0.1$ seconds

Time-step index

$$z^{(1)} = z^{(0)} + h f(t_0, z^{(0)})$$

$$= \begin{bmatrix} 0 \\ 0 \\ 12 \end{bmatrix} + 0.1 \begin{bmatrix} 30 \\ 12 \\ -9.81 \end{bmatrix} = \begin{bmatrix} 3 \\ 1.2 \\ 11.019 \end{bmatrix}$$

$$z^{(2)} = z^{(1)} + h f(t_1, z^{(1)})$$

$$= \begin{bmatrix} 3 \\ 1.2 \\ 11.019 \end{bmatrix} + 0.1 \begin{bmatrix} 30 \\ 11.019 \\ -9.81 \end{bmatrix} = \begin{bmatrix} 6 \\ 2.3019 \\ 10.038 \end{bmatrix}$$

$$z^{(3)} = \text{---} \ldots$$

Matlab Example

$$\frac{dy}{dt} = t - y \qquad y(0) = 1$$

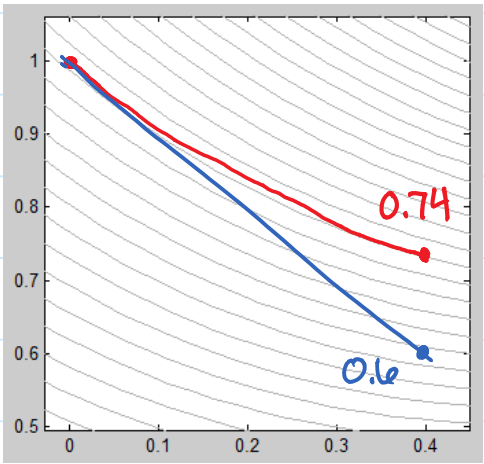
Exact solution is $y(t) = 2e^{-t} + t - 1$

Euler step of length $h = 0.4$

$$y_1 = y_0 + h(t - y_0)$$

$$= 1 + 0.4(0 - 1)$$

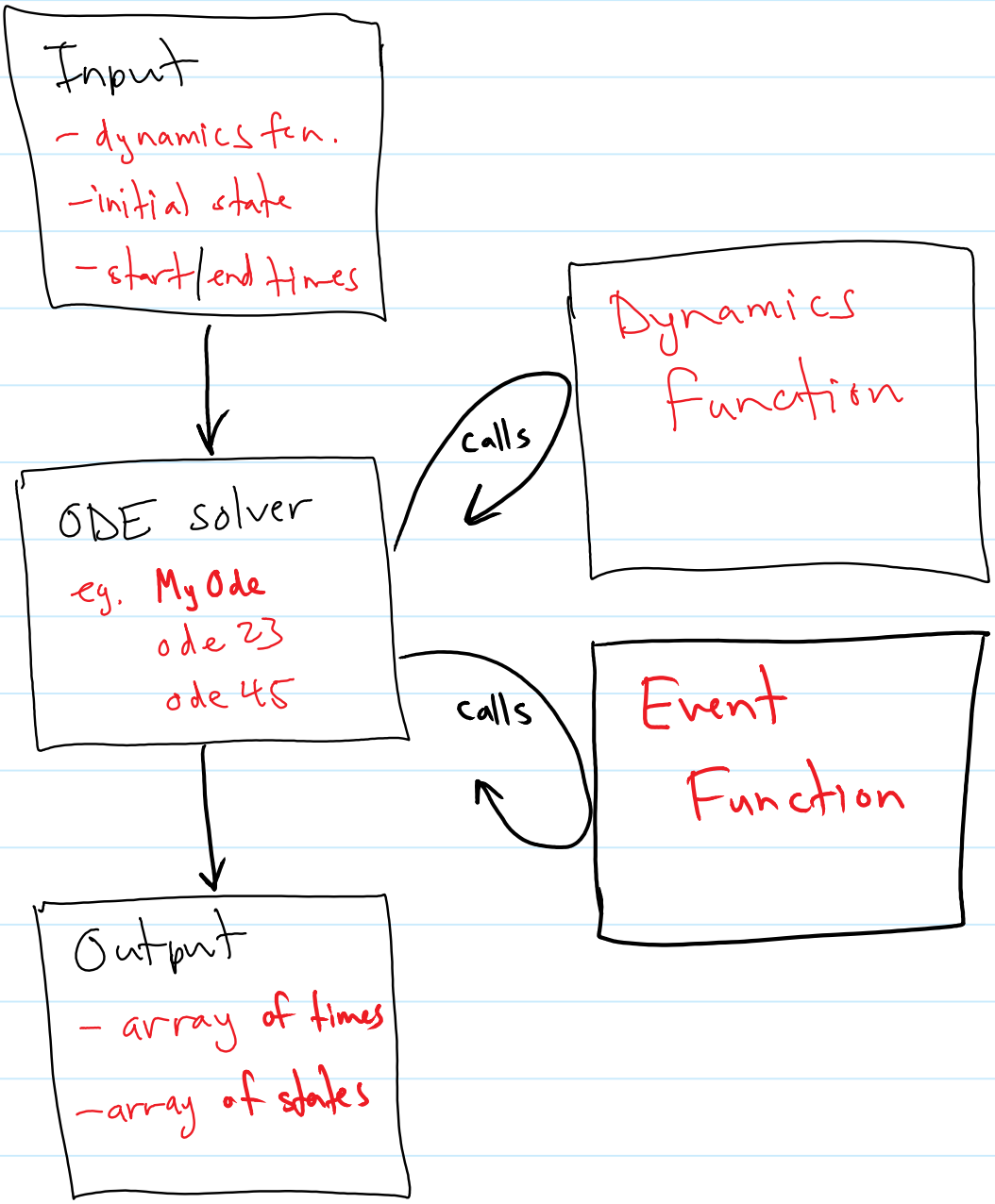
$$= 1 - 0.4 = \boxed{0.6}$$



Exact $\hat{y}_0(0.4) = 2e^{-0.4} + 0.4 - 1 = \boxed{0.7406}$

$\therefore$ local error $l_1 = |0.7406 - 0.6| = 0.1406$

System Architecture for Numerical DE Solvers


Input
- dynamics fcn.
- initial state
- start/end times
Dynamics Function
calls
ODE solver
eg. MyOde
ode23
ode45
calls
Event Function
Output
- array of times
- array of states

Matlab's ODE Suite

Matlab has a bunch of built-in ODE solving routines:

ode45, ode23, ode113, ode155, …

Each has its own strengths and weaknesses. Here's how to use them.

Set up the IVP in standard form

- Create the dynamics function: eg. "simple_de(t,z)"
- Set the initial state: eg. z0 = 1;
- Choose start/end times: eg. tspan = [0 1];

Call the ODE solver

```
[t, y] = ode45(@simple_de, tspan, z0);
```

handle to dynamics fcn; start/end; initial state

(type "help ode45" for documentation)

Interpret output

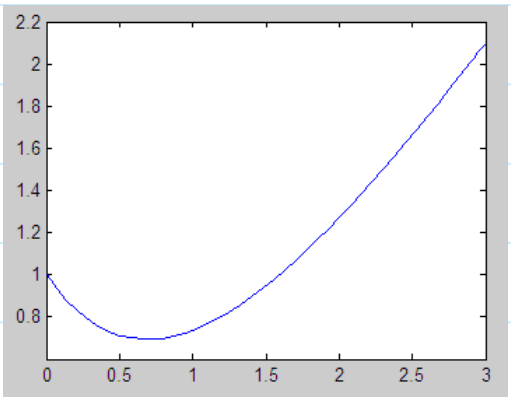
```
plot(t,y);    or    plot(y(:,1), y(:,2));    etc.
```

Example

$$\frac{dy}{dt} = t - y \qquad y(0) = 1$$

```
function dydt = simple_de(t, z)
    dydt = t - z;
```

```
>> [t,y] = ode45(@simple_de, [0 3], 1);
>> plot(t, y);
```



Pass-Through Parameters

Any parameters listed after the 4th parameter in the call to the ODE solver is passed also to the dynamics function.

eg. Recall our previous golf dynamics function

```
function dzdt = simple_golf(t, z)
    % z(1) = x(t)
    % z(2) = y(t)
    % z(3) = y'(t)
    Vx = 30;
    dzdt = [ Vx ; z(3) ; -9.81 ];
```

Vx is "hardcoded"

But we can pass V_x through as a parameter.

```
function dzdt = simple_golf2(t, z, Vx)
    % z(1) = x(t)
    % z(2) = y(t)
    % z(3) = y'(t)
    dzdt = [ Vx ; z(3) ; -9.81 ];
```

Then, to pass V_x from the command where we call the ODE solver...

```
>> [t y] = ode45(@simple_golf2, [0 5], [0;0;20], [], 30);
```

Note → V_x

Events and Options in Matlab's ODE Suite

Before calling the ODE solver, you might need to set up some of the options that govern how the solver behaves.
eg. **events, error tolerances, step sizes, output spacing**

Use the function "odeset" for this.
For example, to specify an events function...

```
>> opts = odeset( 'Events', @MyEvents );
```

Then call the ODE solver with the options structure (stored in "opts").

```
>> [t, y] = ode45(@simple_golf2, [0 5], [0;0;20], opts, 30);
```

dynamics function — start — end — ZO — options — V_x

A full example…

<u>Novelty Golf Driving Range</u>

Dynamics function: golf.m

Events function: golf_events.m

Goal: To learn some numerical ODE methods other than Euler's Method.

Recall Euler's Method

$$y_{n+1} = y_n + h_n f(t_n, y_n)$$

$$l_{n+1} = \left| \hat{y}_n(t_{n+1}) - y_{n+1} \right|$$

Problem: In general, we don't know $\hat{y}_n(t_{n+1})$.

Instead, we approximate it with a higher-order method.

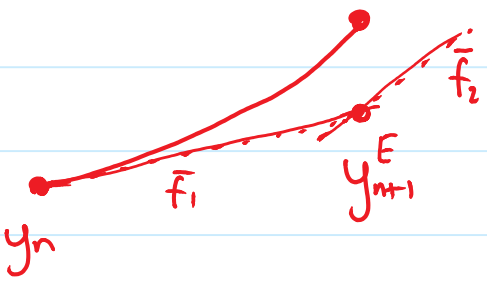
Modified Euler Method

pronounced "rung'-ga"

This is also called "Improved Euler", and "2nd-order Runge-Kutta".

1. Start with an Euler step

$$y_{n+1}^E = y_n + h_n \underbrace{f(t_n, y_n)}_{\bar{f}_1}$$



2. Evaluate f at the new point. ie. get derivatives at (t_{n+1}, y_{n+1}^E)

$$\bar{f}_2 = f(t_{n+1}, y_{n+1}^E)$$

3. Use the average of the two slopes

$$y_{n+1}^M = y_n + h_n \frac{\bar{f}_1 + \bar{f}_2}{2}$$

Modified Euler is a 2nd-order method. Its local error is $O(h^3)$.

The global error of the Modified Euler method is $O(h^2)$.

We can estimate the local error of Euler's Method using

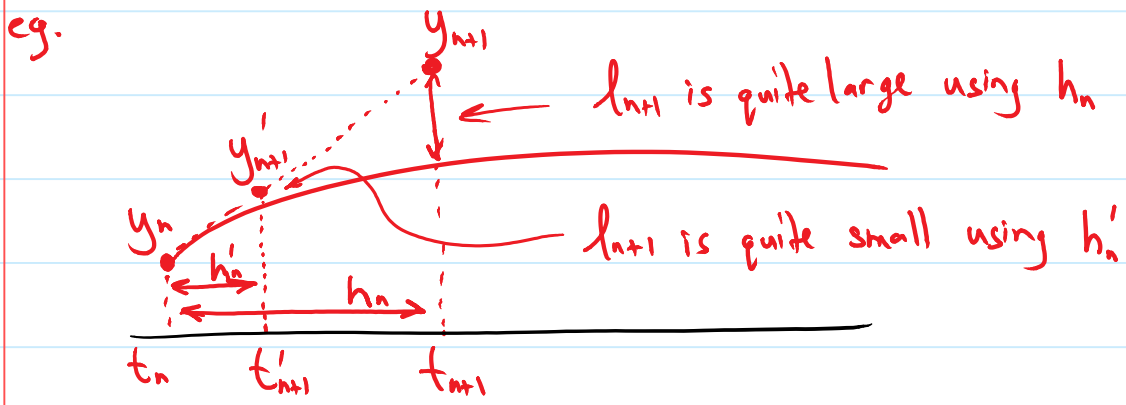
$$l_{n+1}^{E} \approx \left| y_{n+1}^{M} - y_{n+1}^{E} \right|$$

If this combination of methods was implemented into Matlab, it would be called "ode12".

(name based on order of global error).

Adaptive Time-Stepping

We can reduce the local error by simply taking smaller time steps.



However, our method becomes more expensive as we take more and more steps. Thus, choosing a really short time step may not be practical.

We can use our estimate of the local error to choose our time steps.

Pseudocode

1) choose h_n

2) compute y^M_{n+1} & y^E_{n+1}

3) compute l^E_{n+1}

4) if l^E_{n+1} is too big, cut step in half
   - replace h_n with $\frac{1}{2}h_n$
   - go to step (2)

5) else if l^E_{n+1} is really small, take larger steps
   - set $h_{n+1} = 2h_n$ for next step

Matlab's odeset function allows you to set two bounds on the size of the local error.

AbsTol is the maximum allowable local error

i.e. if $|y^M_{n+1} - y^E_{n+1}| >$ AbsTol then shorten the time step & try again.

RelTol is the maximum allowable <u>relative</u> local error

i.e. if $\frac{|y^M_{n+1} - y^E_{n+1}|}{|y^M_{n+1}|} >$ RelTol then shorten time step & try again.

In Matlab, you can set these tolerances using

```
>> opts = odeset('AbsTol', 0.001, 'RelTol', 0.01);
```

Matlab's ODE solvers will accept a time step if it satisfies at least one of AbsTol and RelTol (ie. it does NOT have to satisfy both)

3rd-Order Runge-Kutta

$$\bar{f}_1 = f(t_n, y_n)$$

$$\bar{f}_2 = f\left(t_n + \frac{h_n}{3}, y_n + \frac{h_n}{3}\bar{f}_1\right)$$

$$\bar{f}_3 = f\left(t_n + \frac{2h_n}{3}, y_n + \frac{2}{3}h_n\bar{f}_2\right)$$

$$y_{n+1} = y_n + h_n\left(\frac{1}{4}\bar{f}_1 + 0\bar{f}_2 + \frac{3}{4}\bar{f}_3\right)$$

$\Rightarrow$ ode23 (Matlab demo: rk3.m)

Numerical Stability

Previously, we talked about the convergence of Euler's method.

Let $h \to 0$ and $n \to \infty$ s.t. $hn = \text{constant}$.
Euler's Method converges to $y(x_n)$ with global error $O(h)$.

Now we study the numerical stability.

h fixed, $n \to \infty$
Study how close y_n remains to $y(x)$ for large x.

We study the "test equation",

$$\begin{cases} y' = \lambda y & (\lambda < 0) \\ y(0) = 1 \end{cases}$$

Note: This is a linear ODE, but any nonlinear ODE can be linearized about a chosen point.

Solution is $y(x) = e^{\lambda t}$

Thus, $\lim_{x \to \infty} y(x) = 0$ (since $\lambda < 0$)

Numerical Stability $\Rightarrow$ we require that $\lim_{n \to \infty} y_n = 0$

Stability of Euler's Method

$$\begin{aligned} y_{n+1} &= y_n + h f(x_n, y_n) \\ &= y_n + h\lambda y_n \\ &= (1 + h\lambda) y_n \qquad \therefore y_n = (1+h\lambda)^n y_0 \end{aligned}$$

$$\lim_{n \to \infty} y_n = 0 \iff |1 + h\lambda| < 1$$

$$-1 < 1 + h\lambda < 1$$

$$-2 < h\lambda < 0$$

$-2 < h\lambda < 0$ (ok because $\lambda<0$ & $h>0$)

$\frac{-2}{\lambda} > h$ (recall that $\lambda<0$)

$h < \frac{2}{|\lambda|}$ large $\lambda \rightarrow$ small h

"conditionally stable"

An Implicit Method

The methods we've looked at so far are called "explicit" methods because we have an explicit formula to calculate the next point based on previous points.

An "implicit" method yields an equation involving the next point, as well as the previous point, but must be solved to find out what the next point is.

Trapezoid Method:

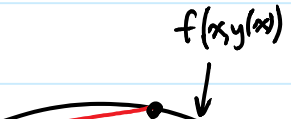
$$y'(x) = f(x,y)$$

Integrate both sides...

$$\int_{x_n}^{x_{n+1}} y'(x)\,dx = \int_{x_n}^{x_{n+1}} f(x,y(x))\,dx$$

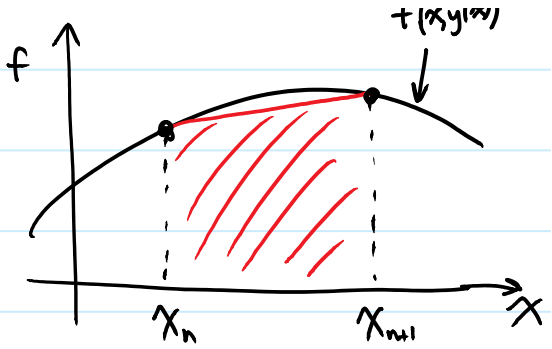
If we know $y(x)$, then $\int_{x_n}^{x_{n+1}} y'(x)\,dx = y(x)\Big|_{x=x_n}^{x_{n+1}} = y(x_{n+1}) - y(x_n)$

How do we approximate $\int_{x_n}^{x_{n+1}} f(x,y(x))\,dx$?



Trapezoid Rule

$$\int_{x_n}^{x_{n+1}} f(x, y(x))\,dx$$



$$\approx \frac{1}{2}\left(f(x_n, y(x_n)) + f(x_{n+1}, y(x_{n+1}))\right) h$$

$$\Rightarrow y_{n+1} - y_n = \frac{h}{2}\left(f(x_n, y_n) + f(x_{n+1}, y_{n+1})\right)$$

- Global error is $O(h^2)$
- To find y_{n+1}, you have to solve a (possibly non-linear) equation
- Implicit methods tend to be more numerically stable than explicit methods.
  - Useful for "stiff" systems

Stability of Trapezoid Method

Again, $y' = \lambda y$, $y(0) = 1$, $\lambda < 0$

$$y_{n+1} = y_n + \frac{h}{2}\left(f(x_n, y_n) + f(x_{n+1}, y_{n+1})\right)$$

$$y_{n+1} = y_n + \frac{h}{2}\left(\lambda y_n + \lambda y_{n+1}\right)$$

$$y_{n+1}\left(1 - \frac{h\lambda}{2}\right) = y_n\left(1 + \frac{h\lambda}{2}\right)$$

$$y_{n+1} = \frac{1 + \frac{h\lambda}{2}}{1 - \frac{h\lambda}{2}}\, y_n$$

Notice that $-1 < \dfrac{1 + \frac{h\lambda}{2}}{1 - \frac{h\lambda}{2}} < 1$ for any h!!

→ "unconditionally stable"
can take larger time steps and still be numerically stable.

Stability of Modified Euler: $h < \frac{2}{|\lambda|}$

Stability of Runge-Kutta 4: $h < \frac{2.785}{|\lambda|}$