

PHY407 – University of Toronto

Lecture 7: Ordinary Differential Equations, part 2/2

Nicolas Grisouard, nicolas.grisouard@utoronto.ca

26 October 2020

Supporting textbook chapters for week 7: Chapters 8.4, 8.2, 8.5.5, 8.6

Lecture 7, topics: * Adaptive step size for RK schemes, * Bulirsch-Stoer method, * Boundary value problems, * Stability issues.

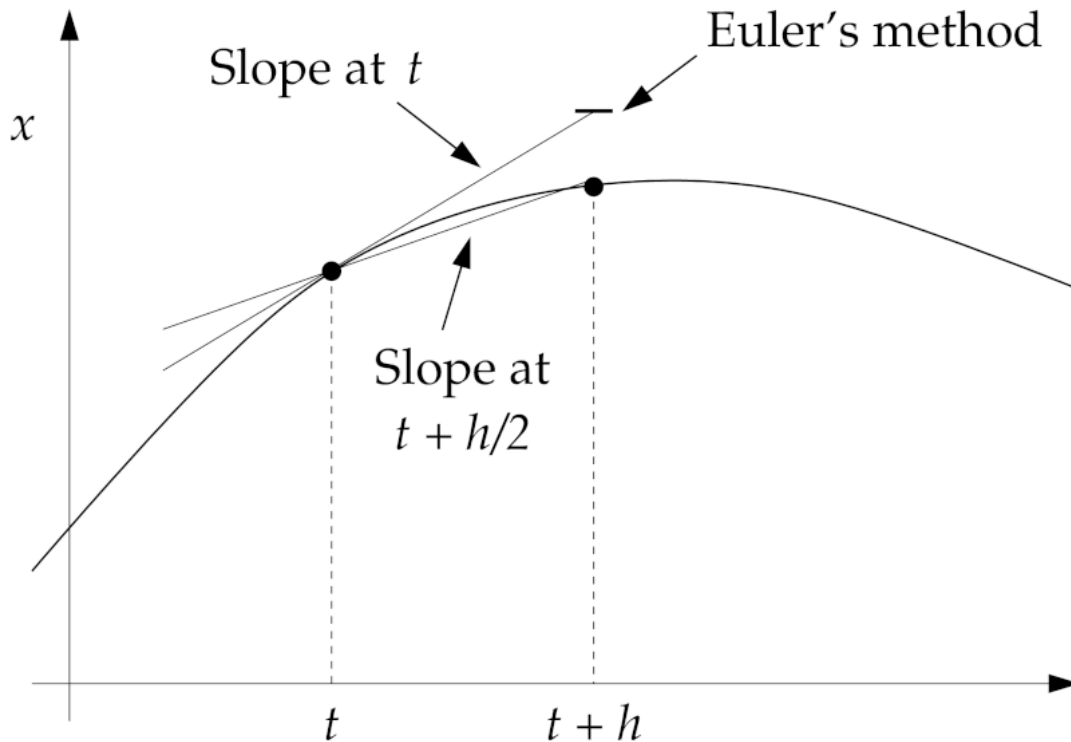
Last week: ODE(s) with some initial condition(s): * 1D: $\frac{dx}{dt} = f(x, t)$ with $x(t = 0) = x_0$. *

nD: $\frac{dx_i}{dt} = f_i(x_1, \dots, x_n, t)$ with $x_i(t = 0) = x_{i0}$. * higher order, e.g.:

$$\frac{d^3x}{dt^3} = f(x, t) \quad \Leftrightarrow \quad \frac{dx}{dt} = v, \quad \frac{dv}{dt} = a, \quad \frac{da}{dt} = f.$$

1 Runge-Kutta methods

1.1 2nd-order Runge-Kutta (RK2) method



RK2: * \oplus Easily(sh) extended to RK4 * \oplus Possible to use adaptive time step (this week) * \ominus time-reversible * \ominus accuracy

RK4: * \oplus accuracy * \oplus Possible to use adaptive time step (this week) * \ominus time-reversible

Leapfrog: * \oplus time-reversible * \oplus basis for higher-order methods (Bulirsch-Stoer, this week) * \ominus accuracy * \ominus time step has to be constant (not exactly true, as we will see).

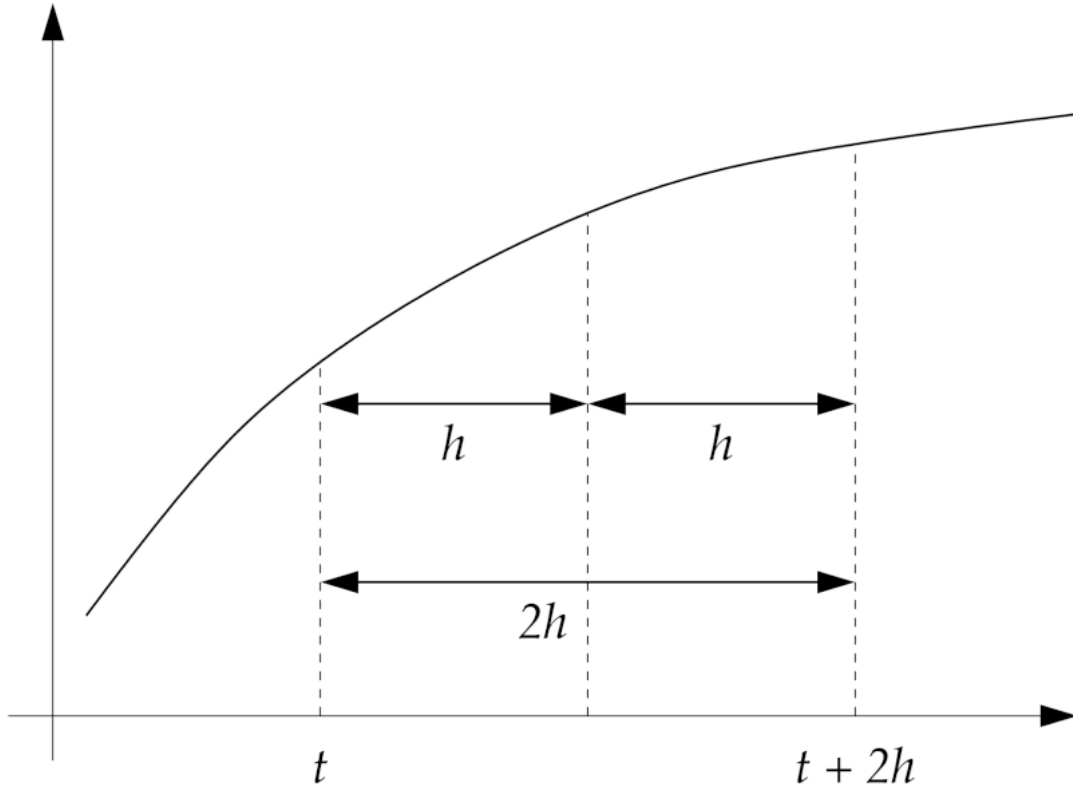
2 Adaptive 4th-order Runge-Kutta method (RK4)

2.1 Error of RK4

- End result:
 - $k_1 = hf(x, t),$
 - $k_2 = hf\left(x + \frac{k_1}{2}, t + \frac{h}{2}\right),$
 - $k_3 = hf\left(x + \frac{k_2}{2}, t + \frac{h}{2}\right),$
 - $k_4 = hf(x + k_3, t + h),$
 - $x(t + h) = x(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4).$
- Very accurate method: error is $\epsilon = ch^5$ at each time step h, c constant (order h^4 globally).
- Very accurate method: error is $\epsilon = ch^5$ at each time step h, c constant.

- Error after 2 time steps? $\approx 2ch^5$.
- Error after 1 time step of $2h$? $c(2h)^5 = 32ch^5 \gg 2ch^5$
- The difference is $(32 - 2)ch^5 = 30ch^5 = 30\epsilon$.
- To estimate error: run ODE solver twice with h (to get x_1), once with $2h$ (to get x_2), divide difference by 30.

$$\epsilon = ch^5 = \frac{1}{30}(x_1 - x_2).$$



2.2 Adaptive time stepping

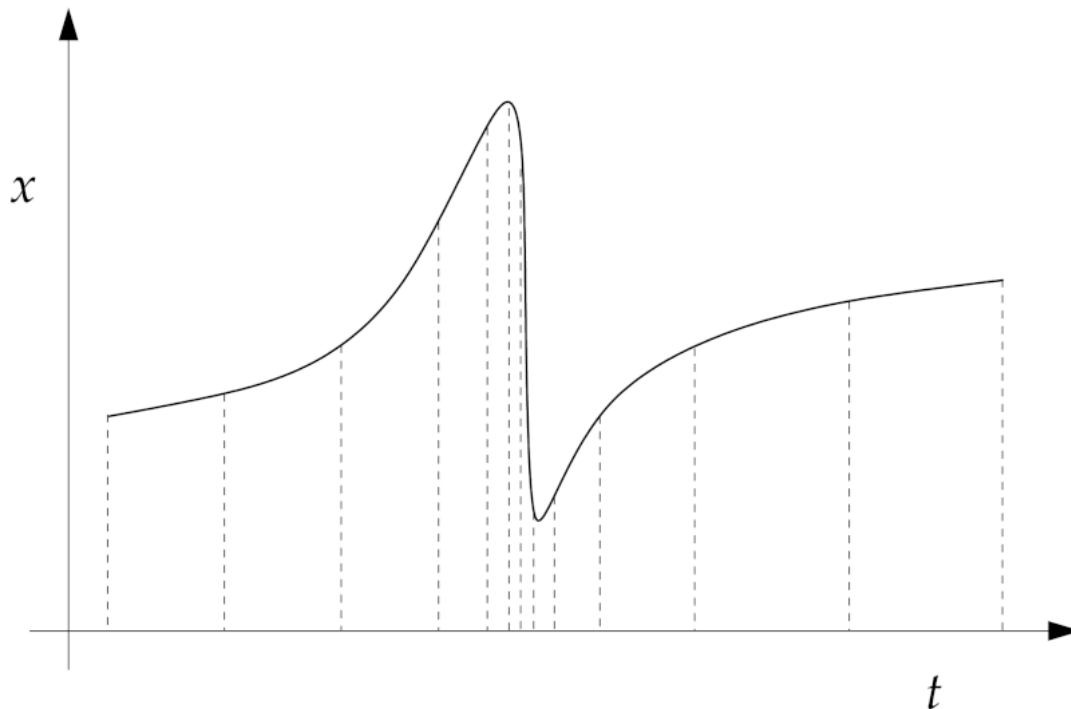
- Suppose target error is δ per unit time (physical time, not step). If

$$\rho = \frac{h\delta}{\epsilon} = \frac{30h\delta}{|x_1 - x_2|} = \frac{30\delta}{ch^4} > 1,$$

h is too small (as in, could be bigger, saving computational resources while still reaching target accuracy) and can be adjusted to $h' = h\rho^{1/4}$ to get $\rho' = 1$.

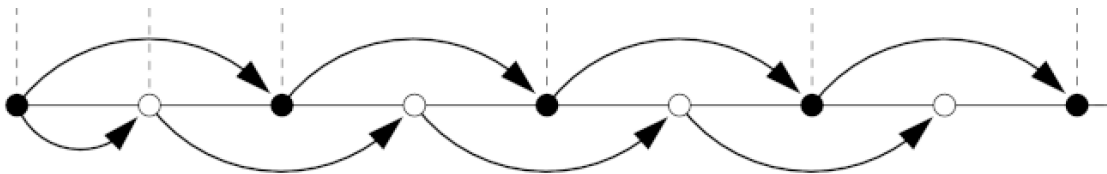
- Still reaches target accuracy.
- Saves calculation time.
- If $\rho < 1$, the time step is too large and needs to be adjusted down by the same factor.
 - We also need to repeat our calculation to get the desired accuracy.
 - This will guarantee meeting error target.
- We test if we need to adjust by performing the calculation twice (we retrieve x_1 and x_2), testing if we met the target, and adjusting h .

- Overall, despite extra work (up to 3 RK4 steps per time step), program often faster because resolution focused where it's needed.



3 From Leapfrog to Bulirsch-Stoer

3.1 Leapfrog error



- Leapfrog is timestep-reversible.
- \Rightarrow error ϵ is an **odd** function of h :

$$\epsilon(-h) = -\epsilon(h)$$

\Rightarrow Taylor expansion is made of **odd** powers of h ,

$$\epsilon(h) = c_3 h^3 + c_5 h^5 + \dots$$

- \Rightarrow cumulative error is **even** in h .
- Each improvement we apply \Rightarrow we can get two orders of accuracy, if we play it right.
- But first, we have to eliminate all even powers in ϵ due to first 1/2 step.

3.2 Modified mid-point (MMP) method

- Even powers of ϵ during the first 1/2 step?
- Integration from t to $t + H$, with $n + 1$ time steps:

$$\begin{aligned}
 x_0 &= x(t) \\
 x_{1/2} &= x_0 + hf(x_0, t)/2 \quad (\text{Initial Euler 1/2-step} \Rightarrow \text{lots of even powers in } \epsilon!) \\
 x_1 &= x_0 + hf(x_{1/2}, t + h/2) \\
 x_{3/2} &= x_{1/2} + hf(x_1, t + h) \\
 &\vdots
 \end{aligned}$$

- ... and keep going until you reach the end.
- So far, this is identical to Leapfrog.
- Then, do **both** the whole integer **and** the forward Euler 1/2-step.

$$\begin{aligned}
 x_{n-1/2} &= x_{n-3/2} + hf(x_{n-1}, t + H - h), \\
 x_n &= x_{n-1} + hf(x_{n-1/2}, t + H - h/2) \approx x(t + H) \\
 x'_n &= x_{n-1/2} + hf(x_n, t + H) \approx x(t + H)
 \end{aligned}$$

- Now do the following adjustment:

$$x(t + H)_{final} = \frac{x_n + x'_n}{2}$$

... and you have canceled the even powers (MMP method).

This is not a trivial result (cf. Gragg 1965 for proof).

3.3 Bulirsch-Stoer method

- MMP method rarely used by itself, but is the basis for the powerful Bulirsch-Stoer method.
- Method:
 - Take 1 single MMP step of size $h_1 = H$ to get estimate of

$$x(t + H) = R_{1,1}.$$

(R stands for “Richardson extrapolation”)

- Now take 2 MMP steps of size $h_2 = H/2$ to get second estimate of

$$x(t + H) = R_{2,1}.$$

- Since we know the MMP has 2nd order and even total error, we can write both of these estimates as

$$\begin{aligned}
 x(t + H) &= R_{1,1} + c_1 h_1^2 + O(h_1^4) \quad \text{and} \\
 x(t + H) &= R_{2,1} + c_1 h_2^2 + O(h_2^4).
 \end{aligned}$$

Recall

$$\begin{aligned}x(t+H) &= R_{1,1} + c_1 h_1^2 + O(h_1^4) \quad \text{and} \\x(t+H) &= R_{2,1} + c_1 h_2^2 + O(h_2^4).\end{aligned}$$

* Using the relationship between the step sizes: $h_1 = 2h_2$, we can equate these expressions to get

$$\begin{aligned}R_{1,1} + 4c_1 h_2^2 + O(h_2^4) &= R_{2,1} + c_1 h_2^2 + O(h_2^4) \\ \Rightarrow c_1 h_2^2 &= \frac{1}{3}(R_{2,1} - R_{1,1}) + O(h_2^4).\end{aligned}$$

* If we plug this back in to the expression for $x(t+H)$ above we get a new estimate called $R_{2,2}$:

$$x(t+H) \approx R_{2,2} = R_{2,1} + \frac{1}{3}(R_{2,1} - R_{1,1}) + \boxed{O(h_2^4)}.$$

$$x(t+H) \approx R_{2,2} = R_{2,1} + \frac{1}{3}(R_{2,1} - R_{1,1}) + \boxed{O(h_2^4)}.$$

* 2 different grid spacings (H and $H/2$) \rightarrow expression for the leading error term \rightarrow replace it with our estimates for these grid spacings, i.e., $R_{1,1}$ and $R_{2,2}$. * We have reduced the error in our estimate by 2 orders! (*which was possible because the errors were even*)

Why stop there?

- Take another grid spacing, to estimate the **new** leading order error term and then replace by that.
- E.g., with $h_3 = H/3$, MMP method yields

$$x(t+H) = R_{3,1} + c_1 h_3^2 + O(h_3^4).$$

- E.g., with $h_3 = H/3$,

$$\begin{aligned}x(t+H) &= R_{3,1} + c_1 h_3^2 + O(h_3^4), \\ &= R_{2,1} + c_1 \left(\frac{3}{2}h_3\right)^2 + O(h_3^4), \\ \Rightarrow R_{3,1} + c_1 h_3^2 + O(h_3^4) &= R_{2,1} + c_1 \frac{9}{4}h_3^2 + O(h_3^4) \\ \Rightarrow c_1 h_3^2 &= \frac{4}{5}(R_{3,1} - R_{2,1}) + O(h_3^4)\end{aligned}$$

- Now plugging this into our expression for $x(t+H)$ and calling it $R_{3,2}$,

$$x(t+H) \approx R_{3,2} + c_2 h_3^4 + \boxed{O(h_3^6)},$$

$$\text{where } R_{3,2} = R_{3,1} + \frac{4}{5}(R_{3,1} - R_{2,1}),$$

and where c_2 is a new constant that we need to find an expression for...

- Equating $R_{3,2}$ and $R_{2,2}$ allows to find c_2 :

$$\begin{aligned} x(t+H) &\approx R_{2,2} + c_2 h_2^4 + \boxed{O(h_2^6)} \\ &\approx R_{3,2} + c_2 h_3^4 + \boxed{O(h_3^6)} \\ h_3 = 2h_2/3 &\Rightarrow c_2 h_3^4 = \frac{16}{65}(R_{3,2} - R_{2,2}) \end{aligned}$$

- Plugging this back in and calling the new result $R_{3,3}$ yields

$$\begin{aligned} x(t+H) &\approx R_{3,3} + O(h_3^6), \\ \text{where } R_{3,3} &= R_{3,2} + \frac{16}{65}(R_{3,2} - R_{2,2}), \end{aligned}$$

- and so on.
- The power in this method is that you keep cancelling 2 powers in the error for every new grid spacing you consider.
- Typically, you continue the refinement until you reach the error tolerance you want.
- Summary of method:
 - Take $h = H$, set $n = 1$ and use MMP to find $x(t+H)$,
 - Continue to refine grid to find new estimates and error estimates.
 - When error is acceptable, stop.
- The iteration can be expressed:

$$\begin{aligned} x(t+H) &= R_{n,m+1} + O(h_n^{2m+2}), \quad \text{where} \\ R_{n,m+1} &= R_{n,m} + \frac{R_{n,m} - R_{n-1,m}}{[n/(n-1)]^{2m} - 1} \quad \text{and} \quad h_n = \left(\frac{n-1}{n}\right) h_{n-1}. \end{aligned}$$

Let's look at `bulirsch.py` from Newman's book.

Equations solved: nonlinear pendulum,

$$\frac{d\theta}{dt} = \omega, \quad \frac{d\omega}{dt} = -\frac{g}{\ell} \sin \theta.$$

Extrapolation table:

$n = 1 : R_{1,1}$				
	\searrow			
$n = 2 : R_{2,1}$	$\rightarrow R_{2,2}$			
	\searrow		\searrow	
$n = 3 : R_{3,1}$	$\rightarrow R_{3,2}$		$\rightarrow R_{3,3}$	
	\searrow		\searrow	\searrow
$n = 4 : \underbrace{R_{4,1}}_{MMP}$	$\rightarrow R_{4,2}$		$\rightarrow R_{4,3}$	$\rightarrow R_{4,4}$

```

[ ]: # load bulirsch.py from Newman
from math import sin, pi

from numpy import empty, array, arange
from pylab import plot, show

g = 9.81
ell = 0.1
theta0 = 179*pi/180

a = 0.0
b = 10.0
N = 100 # Number of "big steps"
H = (b-a)/N # Size of "big steps"
delta = 1e-8 # Required position accuracy per unit time

def f(r):
    theta = r[0]
    omega = r[1]
    ftheta = omega
    fomega = -(g/ell)*sin(theta)
    return array([ftheta, fomega], float)

tpoints = arange(a, b, H)
thetapoints = []
r = array([theta0, 0.0], float)

# Do the "big steps" of size H
for t in tpoints:

    thetapoints.append(r[0])

    # Do one modified midpoint step to get things started
    n = 1
    r1 = r + 0.5*H*f(r)
    r2 = r + H*f(r1)

    # The array R1 stores the first row of the
    # extrapolation table, which contains only the single
    # modified midpoint estimate of the solution at the
    # end of the interval
    R1 = empty([1, 2], float)
    R1[0] = 0.5*(r1 + r2 + 0.5*H*f(r2))

    # Now increase n until the required accuracy is reached

```



```

error = 2*H*delta
while error > H*delta:

    n += 1
    h = H/n

    # Modified midpoint method
    r1 = r + 0.5*h*f(r)
    r2 = r + h*f(r1)
    for i in range(n-1):
        r1 += h*f(r2)
        r2 += h*f(r1)

    # Calculate extrapolation estimates. Arrays R1 and R2
    # hold the two most recent lines of the table
    R2 = R1*1
    R1 = empty([n, 2], float)
    R1[0] = 0.5*(r1 + r2 + 0.5*h*f(r2))
    for m in range(1, n):
        epsilon = (R1[m-1]-R2[m-1])/((n/(n-1))**(2*m)-1)
        R1[m] = R1[m-1] + epsilon
    error = abs(epsilon[0]) # epsilon[0] is theta error

    # Set r equal to the most accurate estimate we have,
    # before moving on to the next big step
    r = R1[n-1]

# Plot the results
plot(tpoints, thetapoints)
plot(tpoints, thetapoints, "b.")
show()

```

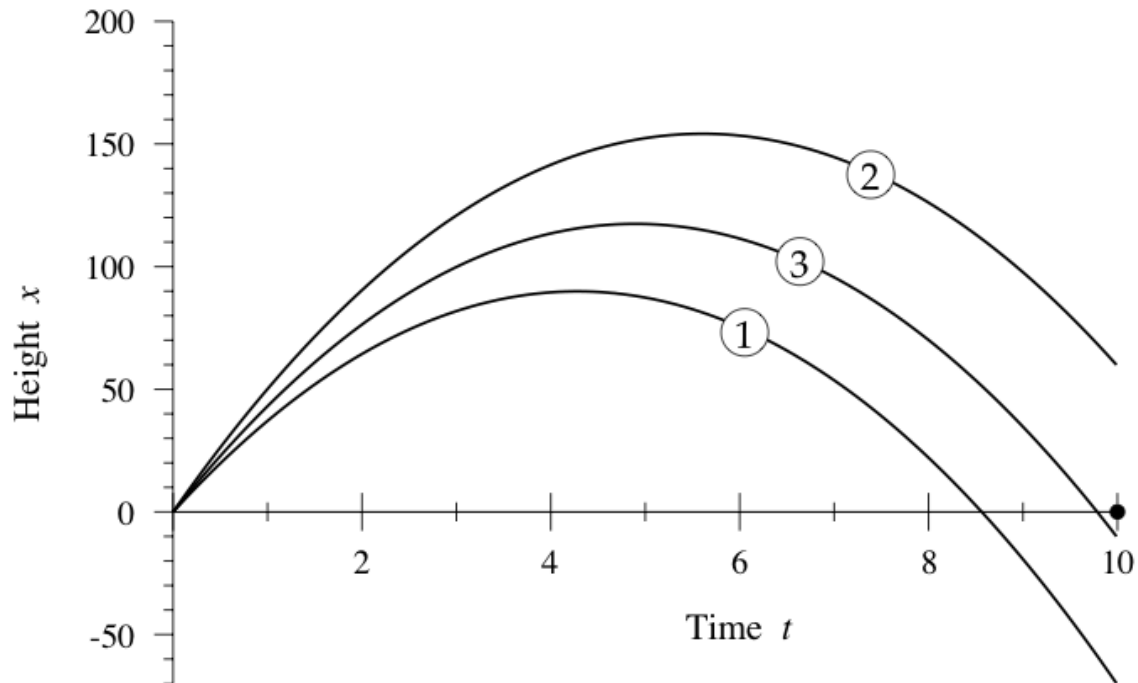
- Notes: This only calculates really accurate values for $x(t + H)$, not the region in between.
- Common practice (helps with efficiency/speed):
 - If your solution doesn't reach your tolerance level in some n_{\max} steps (usually $n_{\max} \sim 8 - 10$), half your interval and redo in smaller H regions.
 - Iterate until your regions are small enough that you reach the tolerance level in n_{\max} steps.
 - "Adaptive" B-S Method!

4 Boundary Value Problems

4.1 Shooting method

- Suppose we wanted to choose an initial velocity v_0 for a projectile to land after $t_l = 10$ s.
- $x(v_0, t)$ is a nonlinear function of v_0 , and finding $x(v_0, t = t_l)$ can be done using root finding method (binary search, secant...)

- **Shooting method:** integrate the equations and adjust v_0 until you locate root.



```
[ ]: # %load throw.py
from numpy import array, arange

g = 9.81 # Acceleration due to gravity
a = 0.0 # Initial time
b = 10.0 # Final time
N = 1000 # Number of Runge-Kutta steps
h = (b-a)/N # Size of Runge-Kutta steps
target = 1e-10 # Target accuracy for binary search

def f(r):
    """Function for Runge-Kutta calculation"""
    x = r[0]
    y = r[1]
    fx = y
    fy = -g
    return array([fx, fy], float)

def height(v):
    """Function to solve the equation and calculate the final height"""
    r = array([0.0, v], float)
    for t in arange(a, b, h):
        k1 = h*f(r)
```

```

        k2 = h*f(r+0.5*k1)
        k3 = h*f(r+0.5*k2)
        k4 = h*f(r+k3)
        r += (k1+2*k2+2*k3+k4)/6
    return r[0]

# Main program performs a binary search
v1 = 0.01
v2 = 1000.0
h1 = height(v1)
h2 = height(v2)

while abs(h2-h1) > target:
    vp = (v1+v2)/2
    hp = height(vp)
    if h1*hp > 0:
        v1 = vp
        h1 = hp
    else:
        v2 = vp
        h2 = hp

v = (v1+v2)/2
print("The required initial velocity is", v, "m/s")

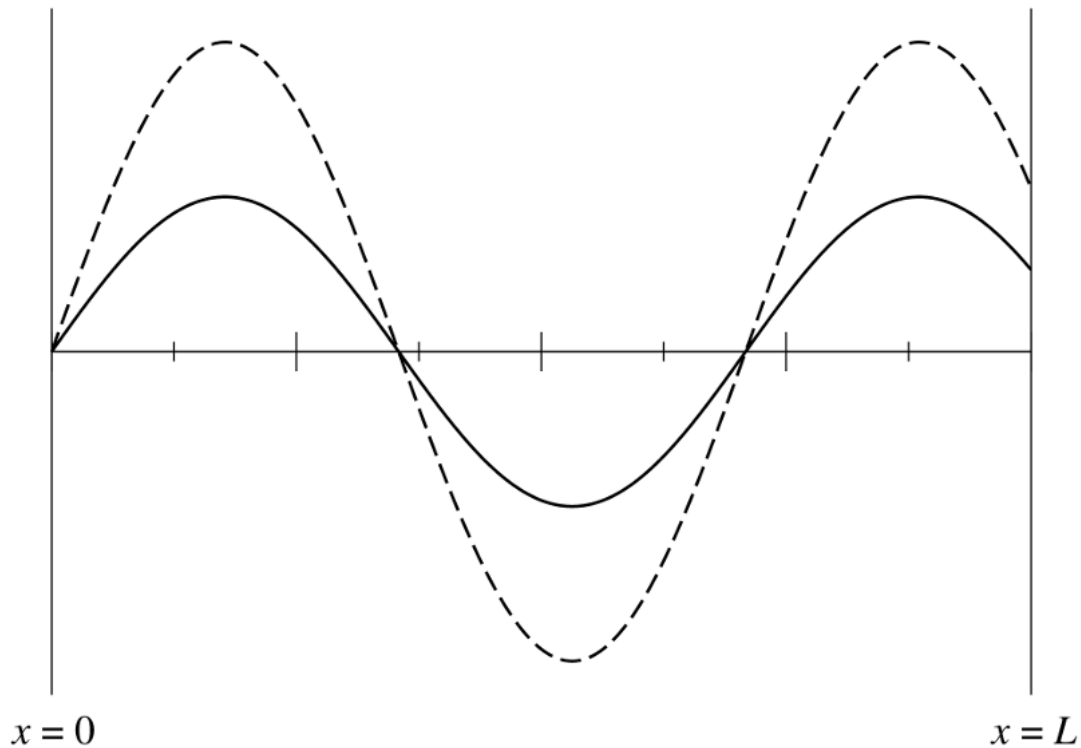
```

4.2 Eigenvalue problems

$$-\frac{\hbar}{2m} \frac{d^2\psi}{dx^2} + V(x)\psi = E\psi,$$

$$\psi(x=0) = \psi(x=L) = 0.$$

- But this approach does not work for finding wavefunctions that satisfy two boundary conditions, as in QM square well, except for valid eigenvalues E .
- So for these problems, E is the parameter that must be varied instead of the leftmost slope of wavefunction.



5 A word on stability

- We have focused on accuracy and speed in investigating our solutions to ODEs.
- But stability is also important. The stability of solutions tells us how fast initially close solutions diverge from each other.
- Some systems are inherently unstable and so will always be challenging to simulate.
- Stability or instability of a system can be determined from small perturbations to a solution of the ODE.
- But even for stable systems, numerical methods can be unstable and give truncation errors that grow.
- Example: $y' = -2.3y$ is a stable system (tends to a finite number). Forward Euler stable for $h = 0.7$ but unstable for $h = 1$.

```
[5]: # vary h in this one
import numpy as np

y0 = 1.
a = 0.
b = 5.
h = 1. # vary this
N = int((b-a)/h)

y = np.empty(N)
```

```

time = np.zeros(N) # I will use this for plotting
y[0] = y0
for k in range(1, N):
    time[k] = k*h
    y[k] = y[k-1] + h*(-2.3*y[k-1]) # Euler time step

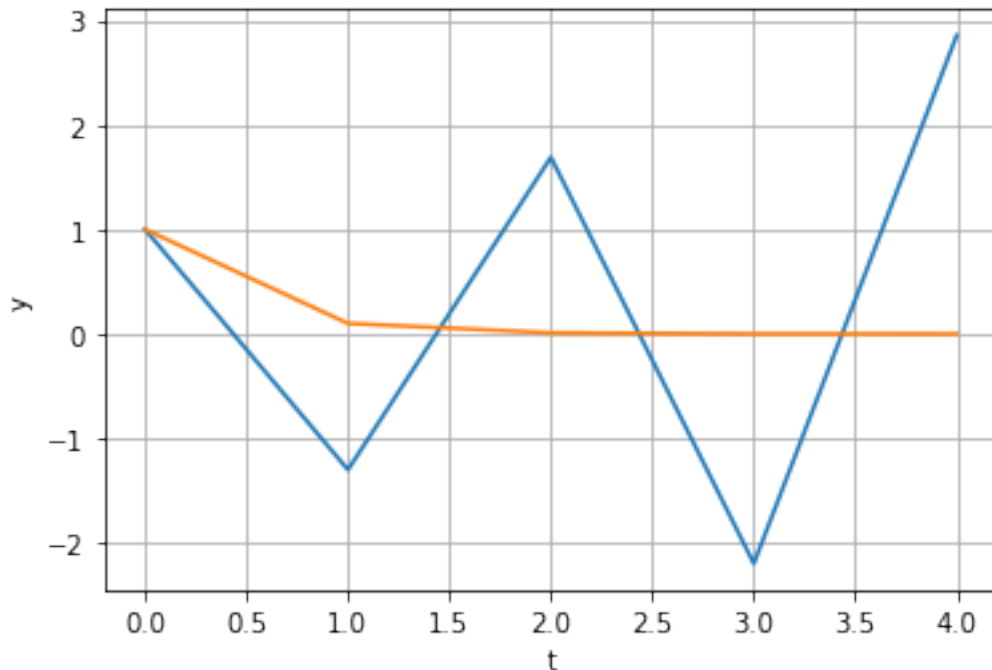
y_a = np.exp(-2.3*time) # analytical solution

```

```

[6]: import matplotlib.pyplot as plt
plt.figure()
plt.plot(time, y, label='Euler, h={0:.1f}'.format(h))
plt.plot(time, y_a, label='Analytical')
plt.xlabel('t')
plt.ylabel('y')
plt.grid()

```



- Why is forward Euler unstable in some cases?
- Explicitly write the solution: for each time step,

$$y_{k+1} = y_k + h_k \lambda y_k \quad (\text{here, } \lambda = -2.3)$$

- And for k time steps,

$$y_k = (1 + h_k \lambda)^k y_0.$$

- For the method to be stable, the magnitude of growth factor

$$|1 + h_k \lambda| \leq 1 \quad \Rightarrow \quad \lambda < 0, \quad h_k \leq |2/\lambda|.$$

- We will investigate more of this in the coming labs.

6 Summary

- Adaptive RK4:
 - Do two RK4 steps, compute the error, adjust step size.
 - More steps “per step”, but gains can be significant if function varies in concentrated regions.
- MMP: a way to make the error even globally (not trivial).
- Bulirsch-Stoer: use MMP to kill even orders of error, one by one.
- BVPs:
 - shooting method: combine ODE integration with root finding,
 - eigenvalue problems: sort of like shooting method, but eigenvalue varies.