Libin K Mathew
Teaching fellow

Libin.mathew@tcd.ie

Reminder

- **Attendance is mandatory**
- **Send an email if you cannot attend explaining the situation**
- **All assignments are mandatory and marked**

EEP5C22 Computational Methods

Beginning with Euler’s method for ODEs


ALTITUDE
AIRSPEED
RISER
G-FORCE
RED BULL STRATOS
FULL POV

Modelling the problem

The Problem

Calculate the velocity of a parachutist $v(t)$ as a function of time t. Assume initial conditions $v(t) = 0$ when $t = 0$.

Newton's 1st Law of Motion: $F = ma = mg - cv(t)$

Hence $$\Rightarrow m\frac{dv(t)}{dt} = mg - cv(t)$$

$$\frac{dv(t)}{dt} = \boxed{\frac{mg - cv(t)}{m}} \qquad (1)$$

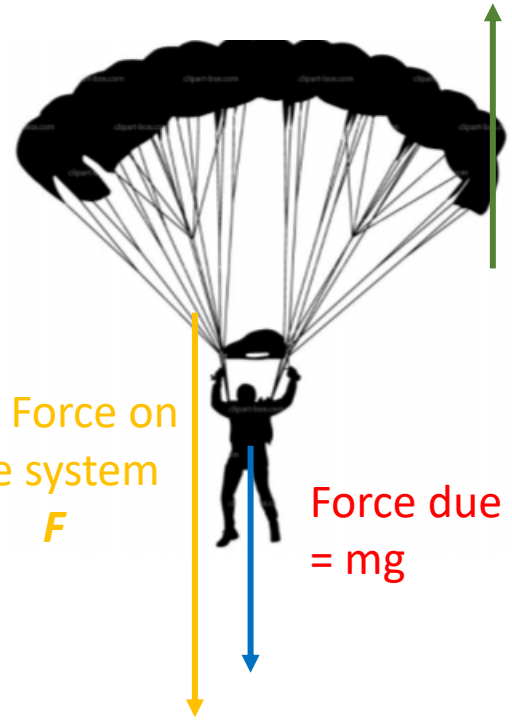


Figure 1: Draw in the forces acting on the system.

We can derive an analytic solution

$$v = \frac{mg}{c}\left[1 - e^{-\left(\frac{ct}{m}\right)}\right]$$

Analytic solution for the velocity of a parachutist

$$m\frac{dv}{dt} = mg - cv(t) \Rightarrow \frac{dv}{g-(c/m)v} = dt$$

$$\int \frac{dv}{g-\frac{cv}{m}} = \int dt + k \tag{2}$$

$$\Rightarrow -\frac{m}{c}\ln\left(g - \frac{cv}{m}\right) = t + k$$

$$\Rightarrow \ln\left(g - \frac{cv}{m}\right) = -\frac{ct}{m} - \frac{ck}{m}$$

$$\Rightarrow \left(g - \frac{cv}{m}\right) = \exp\left(-\frac{ct}{m} - \frac{ck}{m}\right)$$

$$\Rightarrow \frac{cv}{m} = g - \exp\left(-\frac{ct}{m} - \frac{ck}{m}\right)$$

$$\Rightarrow v = -\frac{m}{c}\left[\exp -\left(\frac{ct}{m} + \frac{ck}{m}\right) - g\right] \tag{3}$$

To calculate k, consider initial conditions $v = 0$ at $t = 0$

$$0 = -\frac{m}{c}\left[\exp -\left(\frac{0}{m} + \frac{ck}{m}\right) - g\right]$$

$$\exp -\left(\frac{ck}{m}\right) = g$$

$$\Rightarrow \frac{ck}{m} = -\ln(g)$$

$$\Rightarrow k = -\frac{m}{c}\ln(g)$$

Substituting in equation 3

$$v = -\frac{m}{c}\left[\exp -\left(\frac{ct}{m} + \ln g\right) - g\right]$$

$$v = -\frac{m}{c}\left[g\exp -\left(\frac{ct}{m}\right) - g\right]$$

$$v = \frac{mg}{c}\left[1 - \exp -\left(\frac{ct}{m}\right)\right]$$

From tables

$$\int \frac{dx}{a+bx} = \frac{1}{b}\ln(a+bx)$$

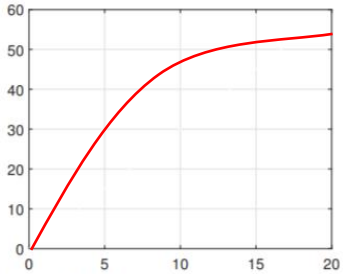


Figure 2: What do you expect $v(t)$ to look like?

Plot the analytic solution using Matlab

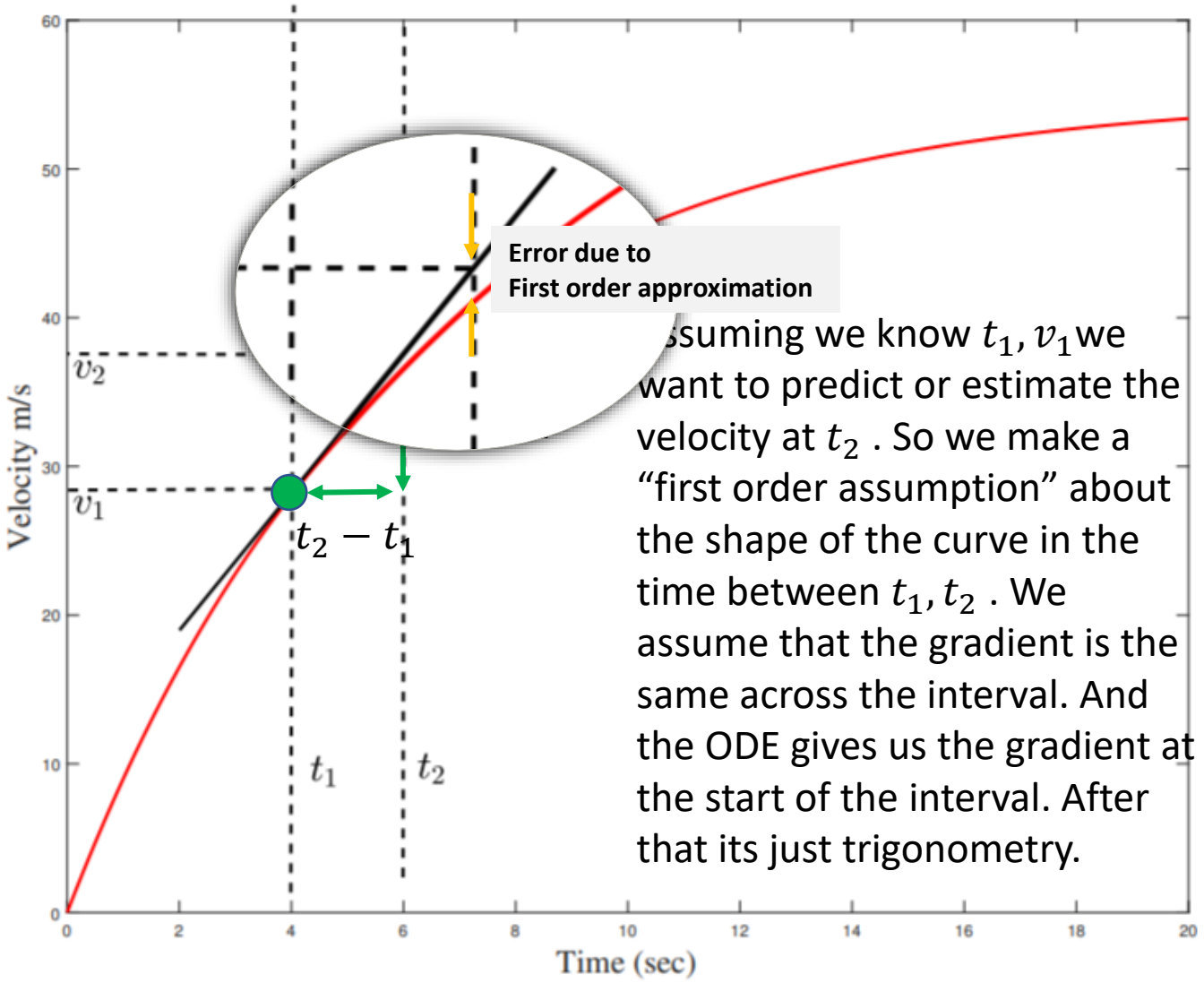
Graphing the solution with Matlab

$$v = \frac{mg}{c}\left[1 - \exp - \left(\frac{ct}{m}\right)\right] \tag{4}$$

```
>> m = 70;
>> g = 9.81;
>> c = 12.5;
>> t = 0 : 0.1 : 20;
>> v = (g*m/c)*(1-exp(-c*t/m));
>> h = plot(t, v, 'r-');
>> set(h, 'linewidth', 2);
>> xlabel('Time (sec)','fontsize',20)
>> ylabel('Velocity m/s','fontsize',20);
```

$$\frac{dv}{dt} = g - \frac{cv}{m}$$

Euler's solution to an ODE



ssuming we know t_1, v_1 we want to predict or estimate the velocity at t_2 . So we make a "first order assumption" about the shape of the curve in the time between t_1, t_2 . We assume that the gradient is the same across the interval. And the ODE gives us the gradient at the start of the interval. After that its just trigonometry.

We show the tangent to $v(t)$ at t_1. In the limit that $t_2 \to t_1$, then $v_2 \to v(t_2)$ Therefore at t_1 the ODE can be written as

$$\frac{v_2 - v_1}{t_2 - t_1} \approx g - \frac{cv_1}{m} \qquad (6)$$

Hence the iterative solution

$$v_2 = v_1 + [g - \frac{cv_1}{m}][t_2 - t_1]$$

and in general

$$v_{n+1} = v_n + [g - \frac{cv_n}{m}][t_{n+1} - t_n]$$

Euler's Solution to 1st Order ODEs

Given $\frac{\partial y}{\partial x} = \phi(x, y)$

Euler proposes $y_{i+1} = y_i + \phi(x_i, y_i)h$

Consider finding the numerical solution to the ODE as follows.

$$\frac{\partial y}{\partial x} = -2x^3 + 12x^2 - 20x + 8.5 \tag{6}$$

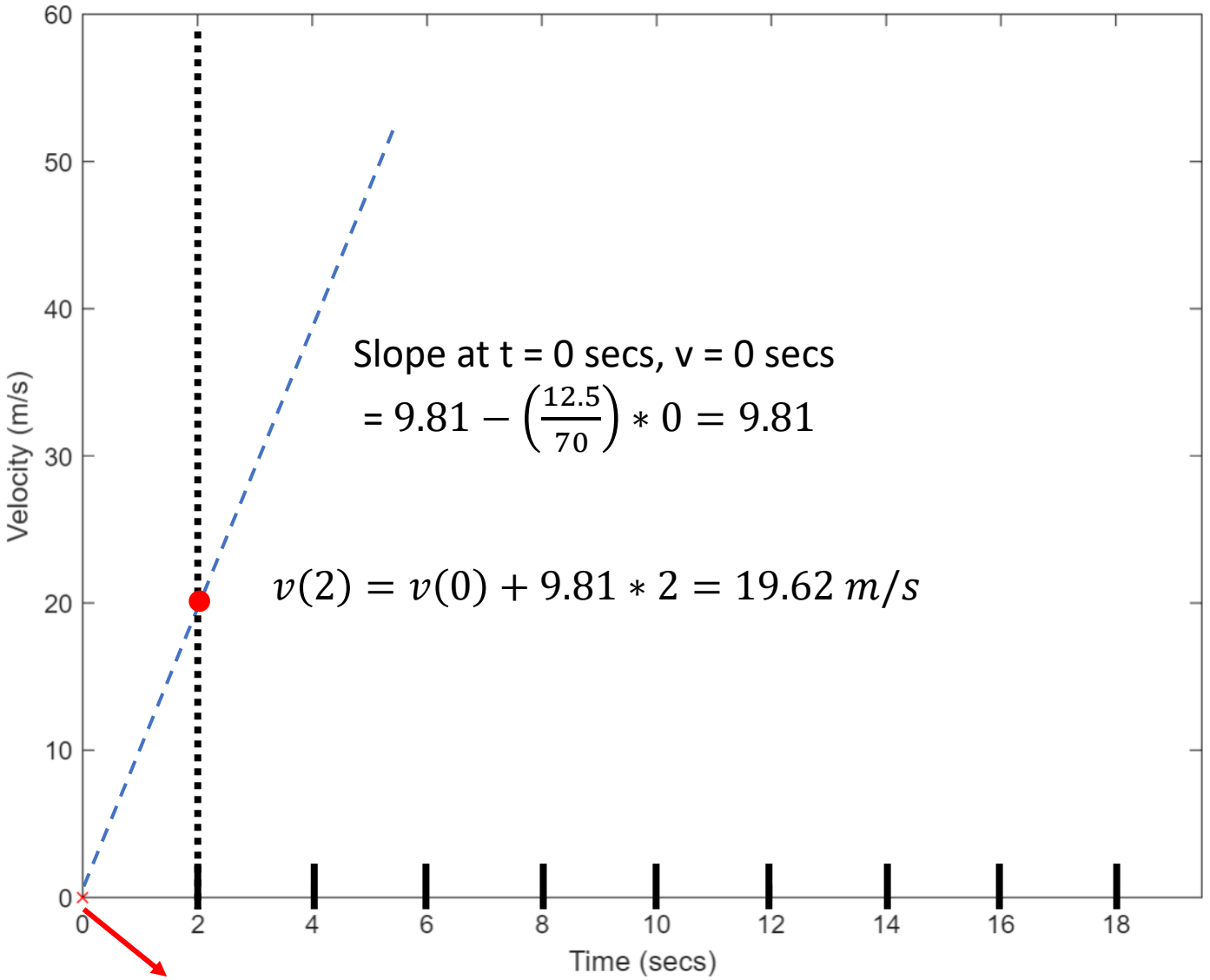
With a step size of h, we can calculate the first derivative at x_i using the ODE itself from equation 6 as follows

$$\phi(x_i) = -2x_i^3 + 12x_i^2 - 20x_i + 8.5 \tag{7}$$

$$v_{n+1} = v_n + [g - \frac{cv_n}{m}][t_{n+1} - t_n]$$

Let's think computationally

$$\frac{dv}{dt} = g - \frac{cv}{m}$$



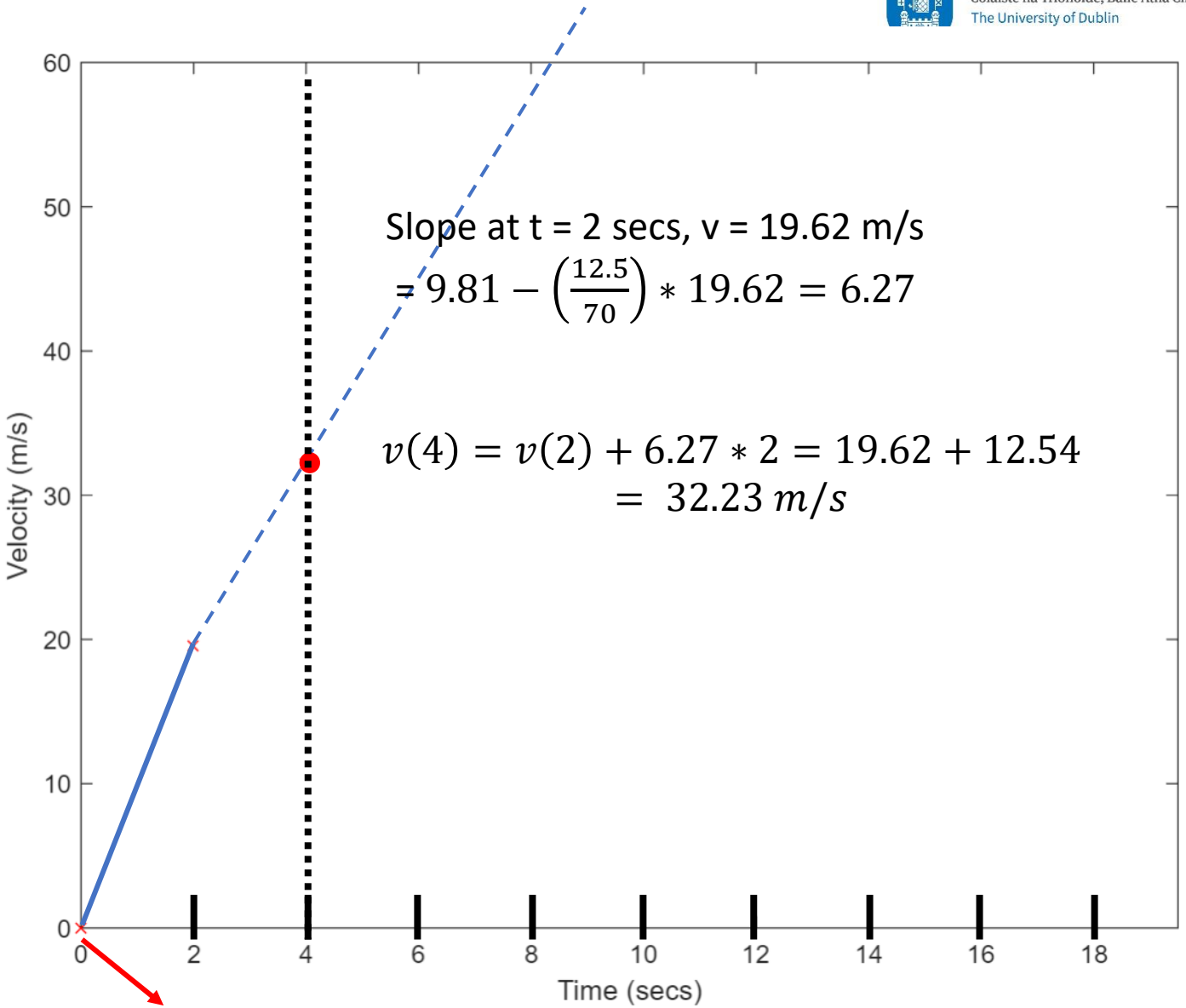
Initial Condition: at t=0, v=0

Let's calculate Euler's solution at t = 2, 4, 6, 8 etc

$$v_{n+1} = v_n + [g - \frac{cv_n}{m}][t_{n+1} - t_n]$$

Let's think computationally

$$\frac{dv}{dt} = g - \frac{cv}{m}$$



Slope at t = 2 secs, v = 19.62 m/s

$= 9.81 - \left(\frac{12.5}{70}\right) * 19.62 = 6.27$

$v(4) = v(2) + 6.27 * 2 = 19.62 + 12.54$
$= 32.23\ m/s$

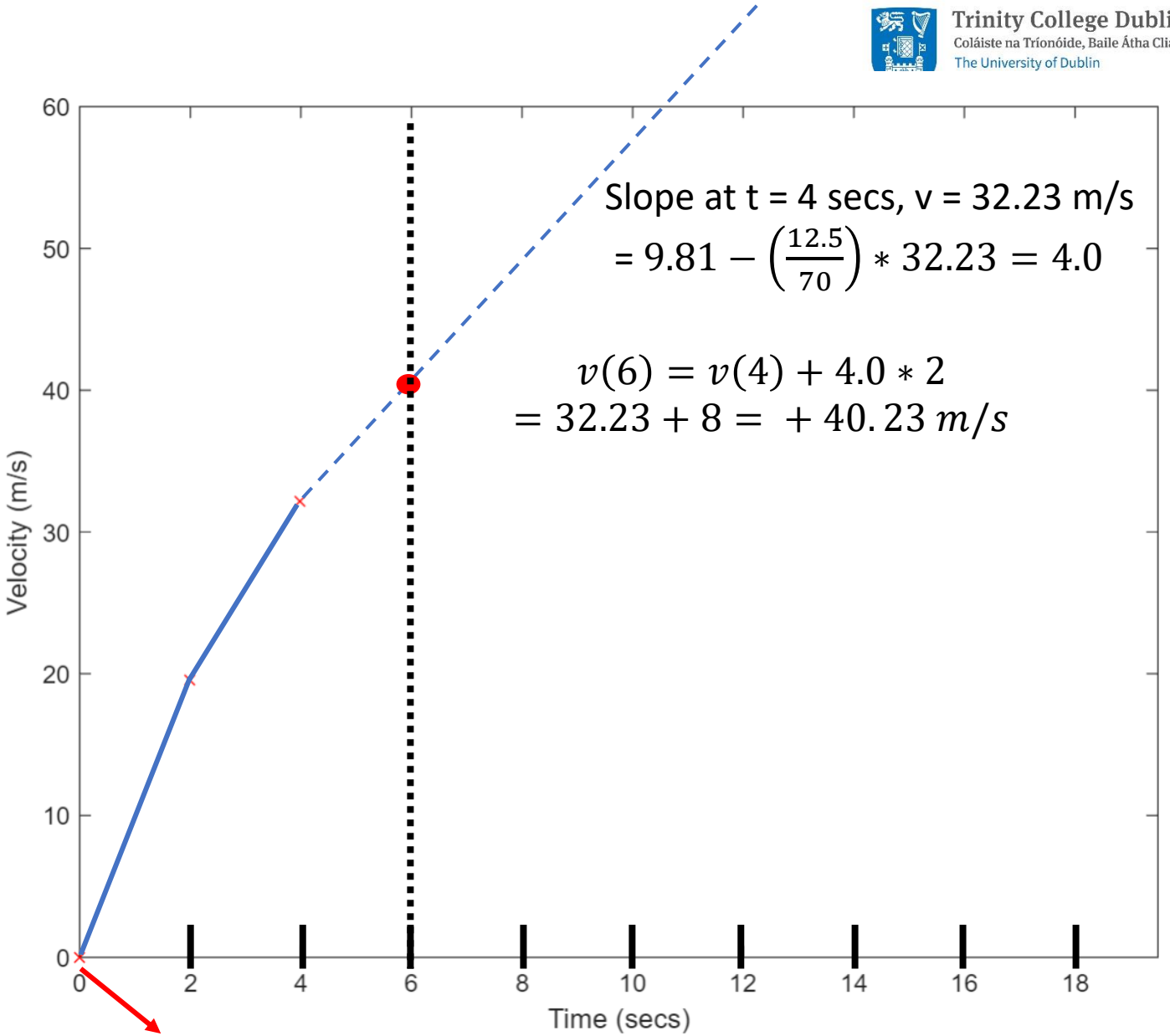
Initial Condition: at t=0, v=0

Let's calculate Euler's solution at t = 2, 4, 6, 8 etc

$$v_{n+1} = v_n + [g - \frac{cv_n}{m}][t_{n+1} - t_n]$$

Let's think computationally

$$\frac{dv}{dt} = g - \frac{cv}{m}$$



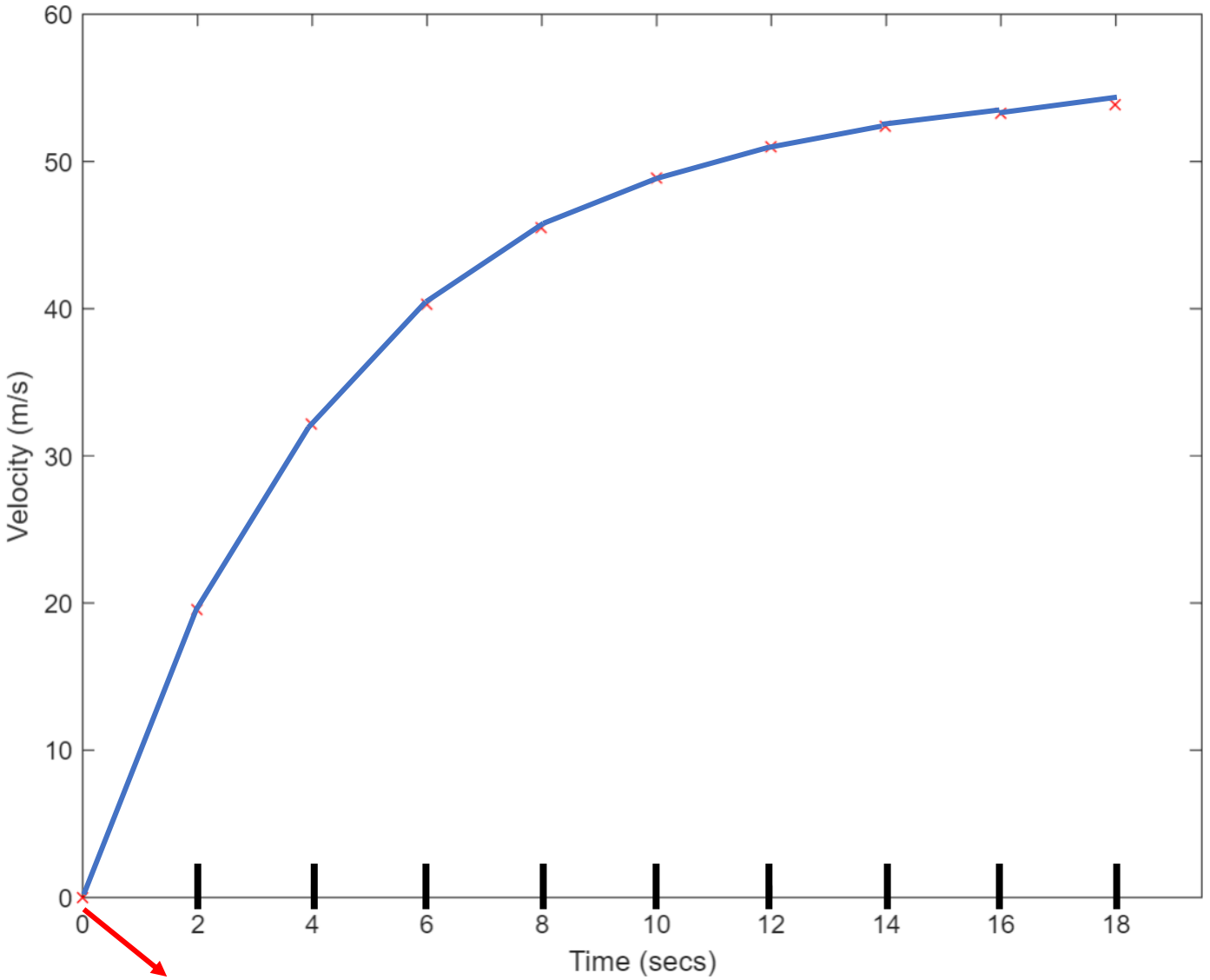
Initial Condition: at t=0, v=0

Let's calculate Euler's solution at t = 2, 4, 6, 8 etc

$$v_{n+1} = v_n + [g - \frac{cv_n}{m}][t_{n+1} - t_n]$$

Let's think computationally

$$\frac{dv}{dt} = g - \frac{cv}{m}$$



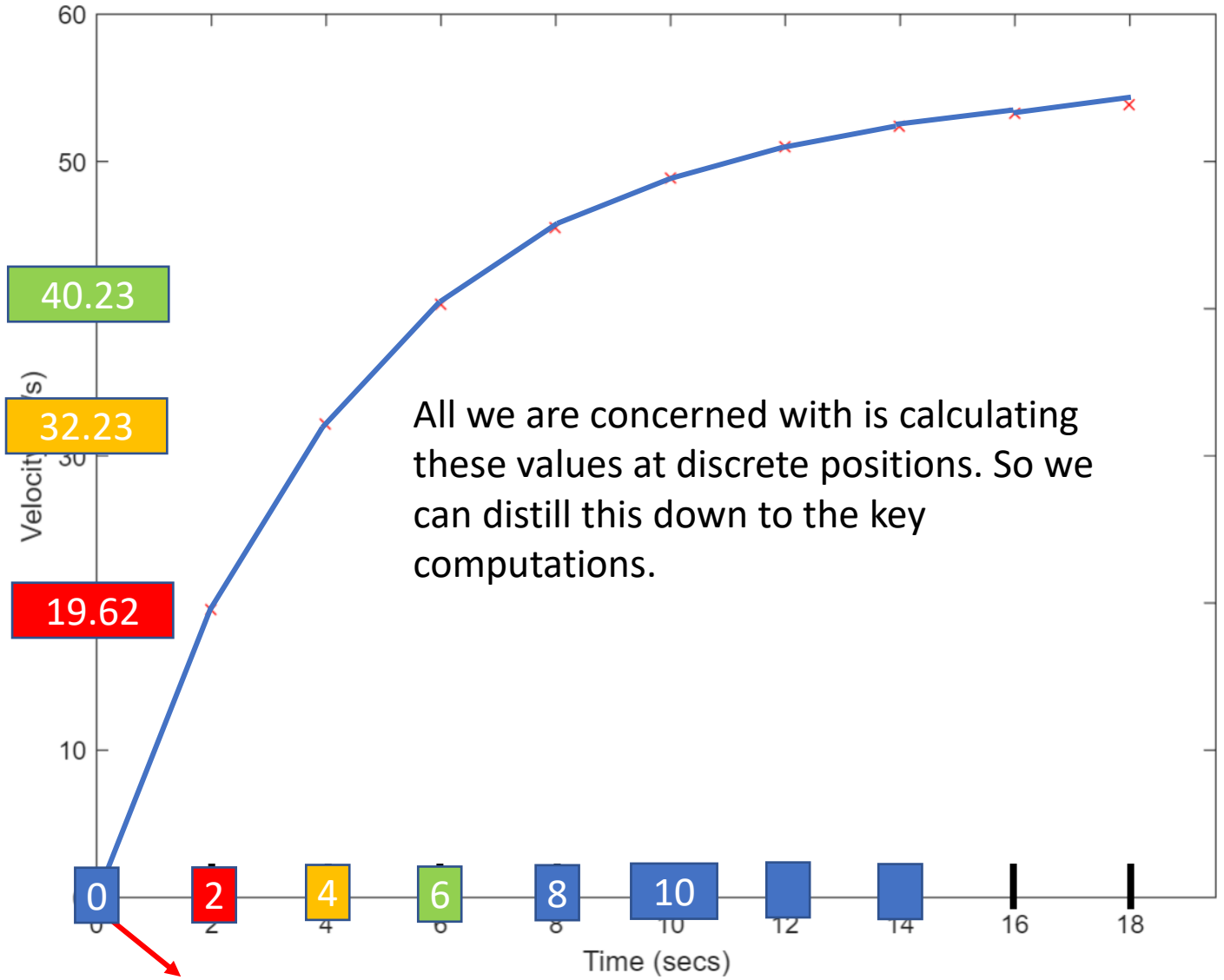
Initial Condition: at t=0, v=0

Let's calculate Euler's solution at t = 2, 4, 6, 8 etc

$$v_{n+1} = v_n + [g - \frac{cv_n}{m}][t_{n+1} - t_n]$$

Let's think computationally

$$\frac{dv}{dt} = g - \frac{cv}{m}$$



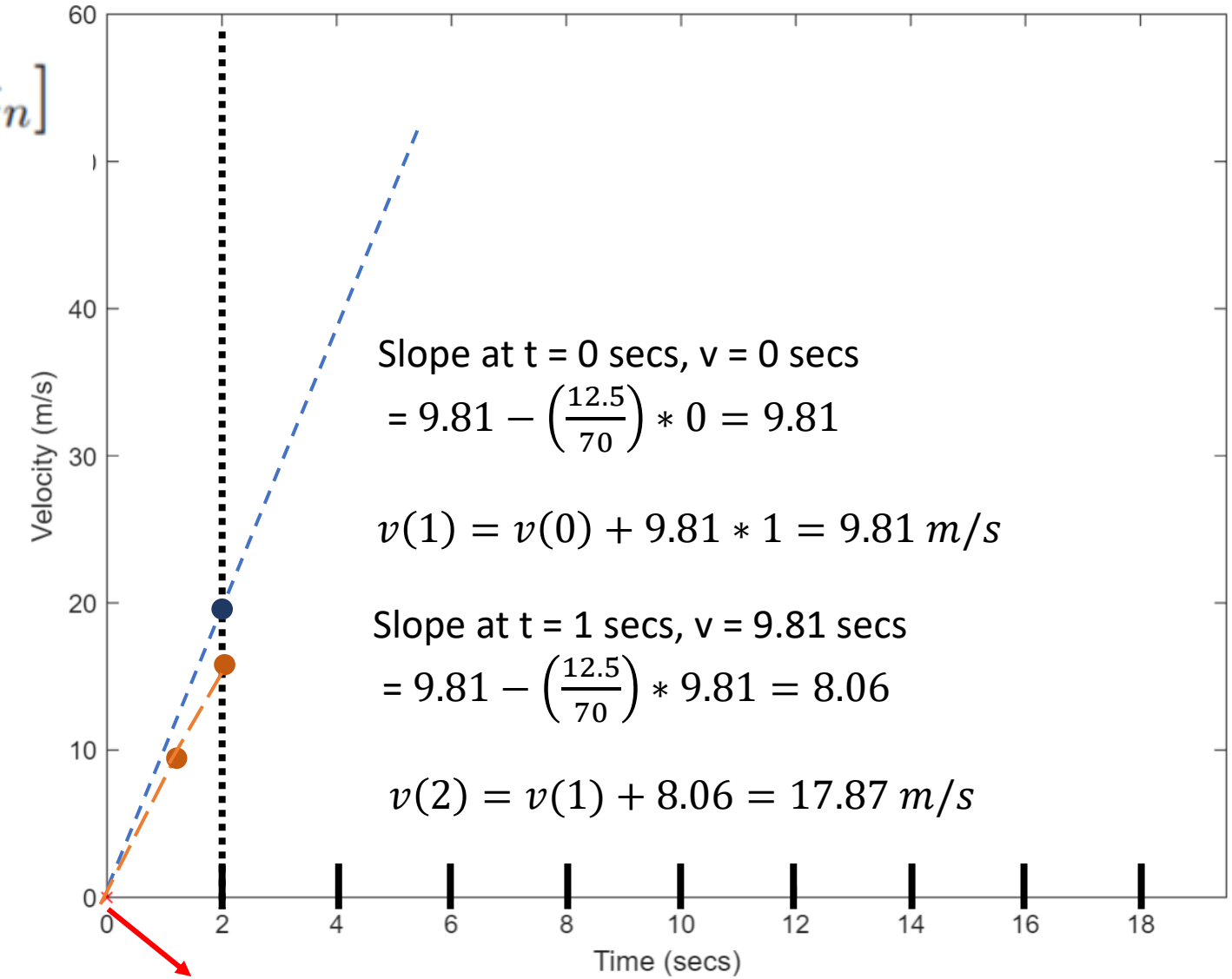
All we are concerned with is calculating these values at discrete positions. So we can distill this down to the key computations.

Initial Condition: at t=0, v=0

$$v_{n+1} = v_n + [g - \frac{cv_n}{m}][t_{n+1} - t_n]$$

What would happen if we assumed a step of 1 instead of 2?

$$\frac{dv}{dt} = g - \frac{cv}{m}$$



Slope at t = 0 secs, v = 0 secs

$= 9.81 - \left(\frac{12.5}{70}\right) * 0 = 9.81$

$v(1) = v(0) + 9.81 * 1 = 9.81\ m/s$

Slope at t = 1 secs, v = 9.81 secs

$= 9.81 - \left(\frac{12.5}{70}\right) * 9.81 = 8.06$

$v(2) = v(1) + 8.06 = 17.87\ m/s$

Initial Condition: at t=0, v=0

Let's calculate Euler's solution at t = 1, 2, 3, etc

Euler's solution gets better with h smaller

Consider finding the numerical solution to the ODE as follows.

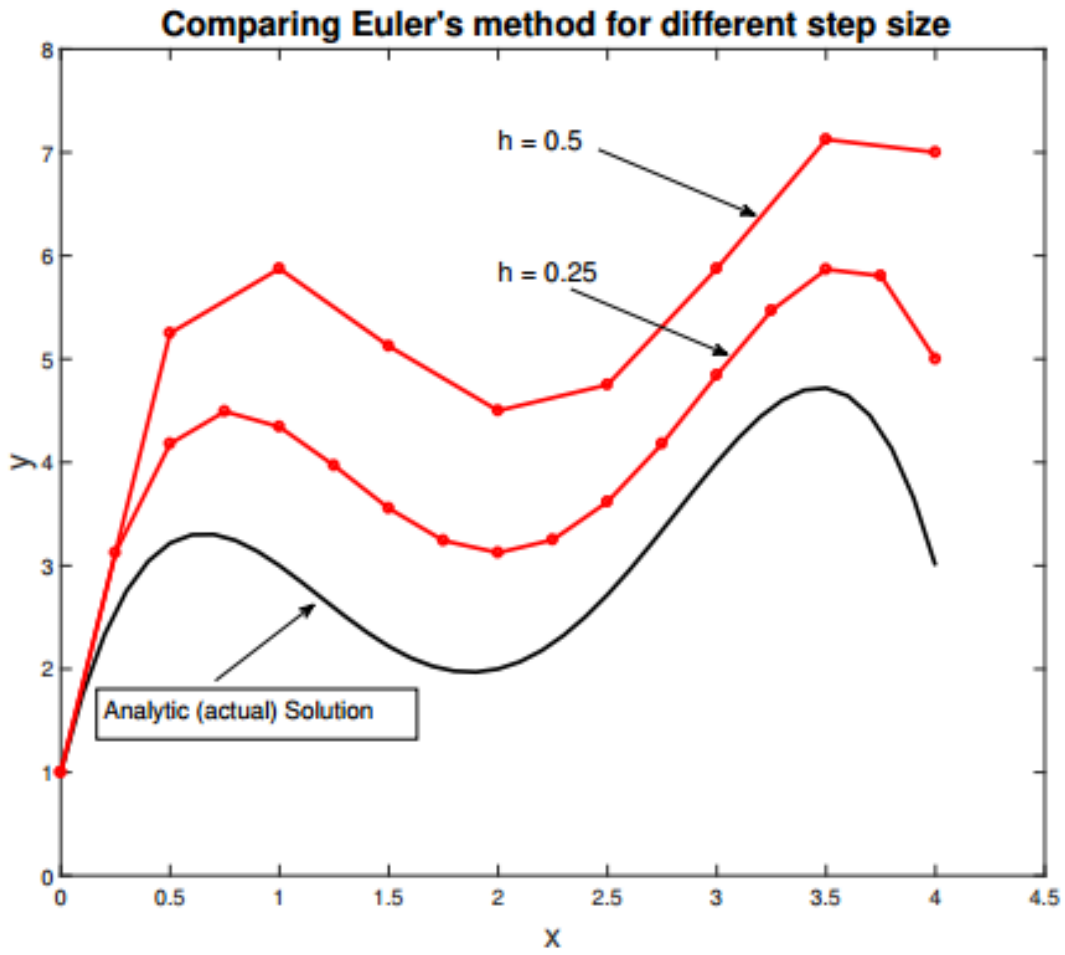
$$\frac{\partial y}{\partial x} = -2x^3 + 12x^2 - 20x + 8.5 \quad (6)$$

With a step size of h, we can calculate the first derivative at x_i using the ODE itself from equation 6 as follows

$$\phi(x_i) = -2x_i^3 + 12x_i^2 - 20x_i + 8.5 \quad (7)$$

Given $\frac{\partial y}{\partial x} = \phi(x, y)$

Euler proposes $y_{i+1} = y_i + \phi(x_i, y_i)h$



We have defined an algorithm: now what about the implementation?

- The algorithm is defined by this equation $$v_{n+1} = v_n + [g - \frac{cv_n}{m}][t_{n+1} - t_n]$$
- It is iterative (might even say recursive)
- It computes values of velocity iteratively based on previous estimates of velocity (*Euler's method is a predictor method btw*).
- But the algorithm can be implemented in different ways
  - Software
  - Hardware
  - Different languages and different platforms

Lab 2 - Your first computational methods exercise

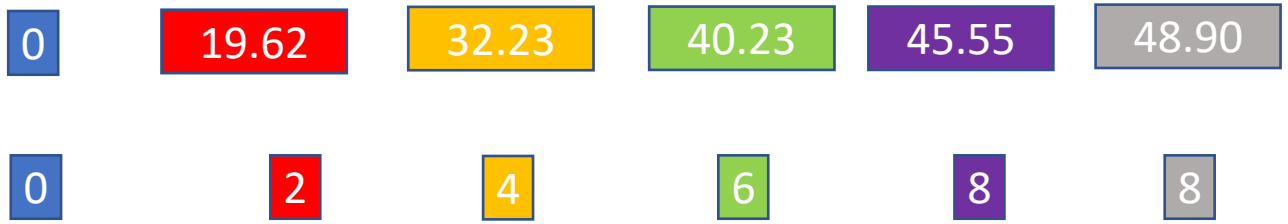
- Write a program script in Matlab that calculates Euler's solution to a first order differential equation (see the exercise description for details)
- Calculate the largest step size h for which the error between Euler's solution and the analytic solution is no larger than 5% of the analytic solution at any time over the given interval (see the exercise description for details)

This time you will submit your code (.m file(s)) separately from the online copy in the embedded matlab on Blackboard.

Let's think computationally and come up with an implementation

$$\frac{dv}{dt} = g - \frac{cv}{m}$$

$$v_{n+1} = v_n + [g - \frac{cv_n}{m}][t_{n+1} - t_n]$$



v_1	v_2	v_3	v_4	v_5
t_1	t_2	t_3	t_4	t_5

We are calculating an array of values v_n at each timestamp t_n . In a sense we are successively stepping over values in another array t_n calculating the velocities as we go. Because the time step is the same every time, all we have to know is that we are stepping from one element to the next one in the array. **We don't have to know the actual time at each time stamp. All we have to know is the initial condition i.e. the first element in each array.**

Let's think computationally and come up with an implementation

v_1	v_2	v_3	v_4	v_5
t_1	t_2	t_3	t_4	t_5

We are calculating an array of values v_n at each timestamp t_n

As soon as you start to think about implementing this you realise that you are "stepping" over elements in an array. Or "using" consecutive elements in an array. Or you get the next value by processing the last one.

So that implies using a "for" loop to step over the independent variable. (Or a "while" loop)

You know in this case that your ultimate goal is to populate an array "v", say with values one element at a time.

So you should be thinking that I need to create a loop that visits every "box" in that array one at a time.

$$v_{n+1} = v_n + [g - \frac{cv_n}{m}][t_{n+1} - t_n]$$

Matlab example for Euler's ODE Method

$$v_{n+1} = v_n + [g - \frac{cv_n}{m}][t_{n+1} - t_n]$$

```
h = 0.5;
% Set up all my increments in x
x = 0 : h : 4.0;
n = length(x);
y = zeros(1, n);
% Set initial condition (y = 1 when x = 0)
y(1) = 1;
% This is Euler
for i = 1 : n - 1,
  slope = differential1(x(i));
  y(i+1) = y(i) + slope * h;
end;
```

Discuss Lab1

Reference Solution

```
% Cone parameters
r = 5;
h = 12;

vol = pi * r^2 *h / 3;

% Calculate the volume vector
r_vals = [0, 1, 3.3, 4, 5];
volumes = pi * r_vals.^2 * h / 3;

figure(1);
graph_properties = plot(r_vals, volumes, 'r-*');
set(graph_properties, ‘LineWidth', 3.0);
% or graph_properties.LineWidth = '3’;
xlabel('Radius', 'fontsize', 20);
ylabel('Volume', 'fontsize', 20);
shg;
```

Linting

Lint, or a linter, is a static code analysis tool used to flag programming errors, bugs, stylistic errors, and suspicious constructs.

What happens, if we would like to merge code from different developers, A and B?

Developer A

```
% Cone parameters
r = 5;
h = 12;

vol = pi * r^2 * h / 3;

% Calculate the volume vector
r_vals = [0, 1, 3.3, 4, 5];
volumes = pi * r_vals.^2 * h / 3;

figure(1);
graph_properties = plot(r_vals, volumes, 'r-*');
graph_properties.LineWidth = '3’;
xlabel('Radius', 'fontsize', 20);
ylabel('Volume', 'fontsize', 20);
shg;
```

Developer B

```
radius=5;height=12;
vol=(pi*radius*radius*height)/3;
  radius_vals=[0,1,3.3,4,5];
 volumes = pi* radius_vals.^2 * h /3;

figure(1);graph_properties = plot(r_vals, volumes, 'r-*’);
    graph_properties.LineWidth = '3’;
xlabel('Radius', 'fontsize', 20);
ylabel('Volume', 'fontsize', 20);shg;
```

Matlab Code Styling / Linting

- Scripts are allowed only if the first line of the script starts with **clear** and **close all**.
- Do not use figure without an accompanying argument, e.g. figure; instead use figure(1)
- Do not overload Matlab keywords, e.g. sum = 0;
- Line Length is maximum 80 characters. Use ... to break lines, e.g.

```
s = ['A very long and extremely boring string that must be ' ...
    'split in two pieces.'];
```

- One statement per line, e.g. not clear; close all;

Matlab Code Styling / Linting

- **Indentation**: Use 2 chars of whitespace to indent code blocks and nested functions

```
for p = 1:100,
  fprintf('p is %f\n',p);
  m = p/2;
  fprintf('m is %f\n',m);
  if (m > 20)
    fprintf('m is bigger than 20\n');
  end;
end;
```

The following is NOT allowed.

```
for p = 1:100,
fprintf('p is %f\n',p);
m = p/2;
fprintf('m is %f\n',m);
if (m > 20)
fprintf('m is bigger than 20\n');
end;
end;
```

- **Whitespacing**: Use one space around operators but not : ;) (

```
x = x + 1;
y(1:10) = Spam([1, 2, 3, 4]);
q = ~p;
```

The following is NOT allowed

```
x = x+1;
y(1 : 10)=Spam([1,2,3,4]);
q=~ p;
```

Matlab Code Styling / Linting

- **Naming**:
  - ✓ Use descriptive words for variables, snake_case (separated by _).
  - ✓ Use camel case for function names.

- **Blank** lines: Use two blank lines between function definitions.

- **Comments**: wherever the rationale for a line of code is not obvious. Use comments to explain what a function does, what a loop does, why you use specific values etc

camelCase

kebab-case

snake_case

```
computed_velocity = pi * 16.4 - c * velocity;
new_velocity = dragCalculation(computed_velocity);
```

The following is not allowed.

```
computedvelocity = pi * 16.4 - c * velocity;
newVelocity = drag_Calc(computedvelocity);
```

Matlab Code Styling / Linting

- **Comments**: wherever the rationale for a line of code is not obvious. Use comments to explain what a function does, what a loop does, why you use specific values etc

```
function [bark, woof] = DogSong(breed, pet, pat, good_dog, sit, scratch);
% DogSong -- Sing like a dog.
%
% [bark, woof] = DogSong(breed, drink, pat, good_dog, sit, scratch)
% Simulates the sequence of barks and woofs produced by a dog of
% given breed under various owner-interaction scenarios.
%
% Arguments
%
% breed: Character string identifying the dog's breed.
% pet: Optional. A 1x3 double array of petting components. Specify
%      the empty array [] to use the default value [0, 0, 0].
% pat: Logical scalar, indicating whether the dog is being patted on
%      the head
% good_dog: A 1xN cell array of character strings, each containing
%           an appreciative sentence for the dog's behavior.
% sit: Optional. Logical scalar indicating whether the dog should
%      be sitting. Default is true.
% scratch: Optional. Logical scalar indicating whether the dog is
%          allowed to scratch while singing. Default is false
%
% Return values
%
% bark: An MxN double array of barf spectra, each row representing
%       the spectral distribution in the human-audible range of one
%       dog utterance.
% woof: An Mx1 cell array of character strings, each containing a
%       human-readable rendition of the corresponding row in barf.
%
% Required toolboxes: None.
```