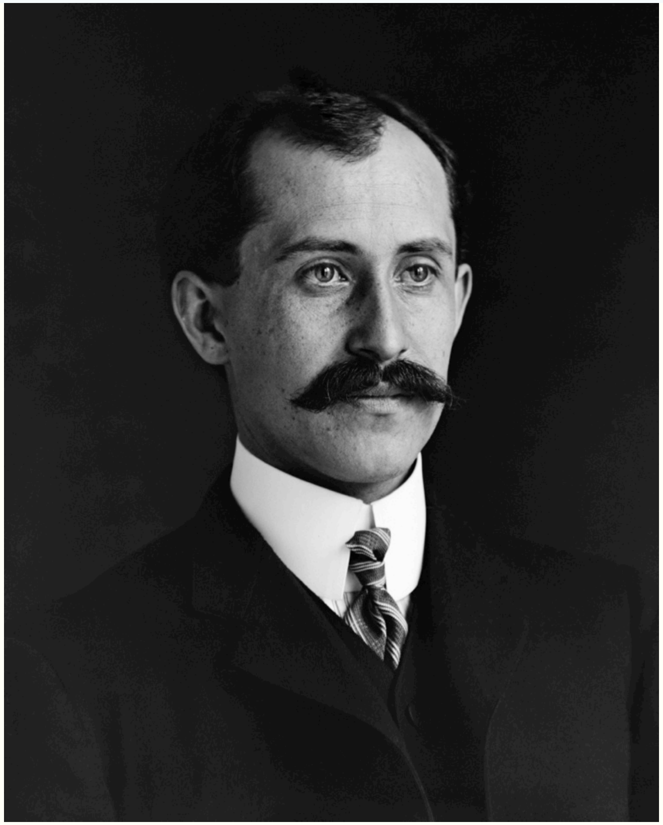
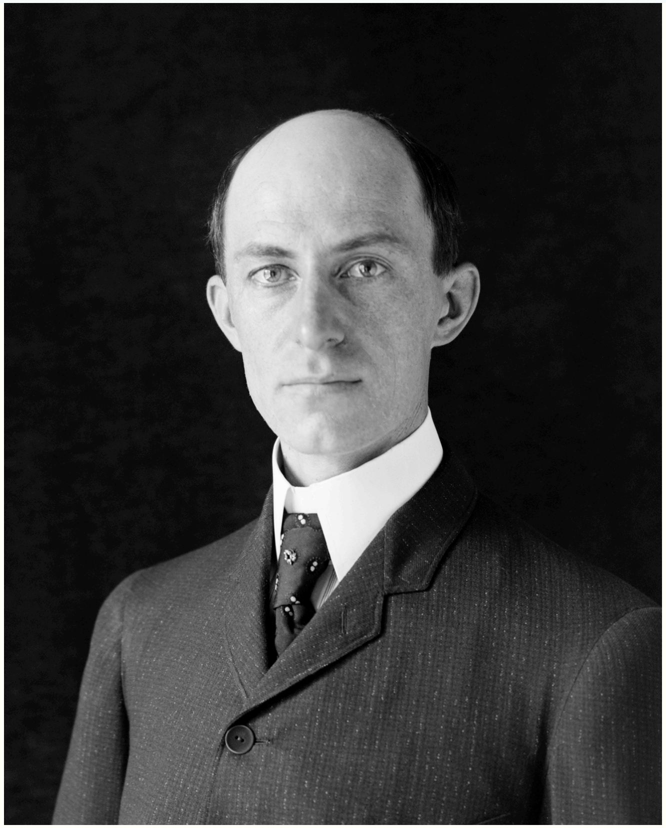
ENGG1811 Computing for Engineers

Week 7A: Simulation

Wright brothers





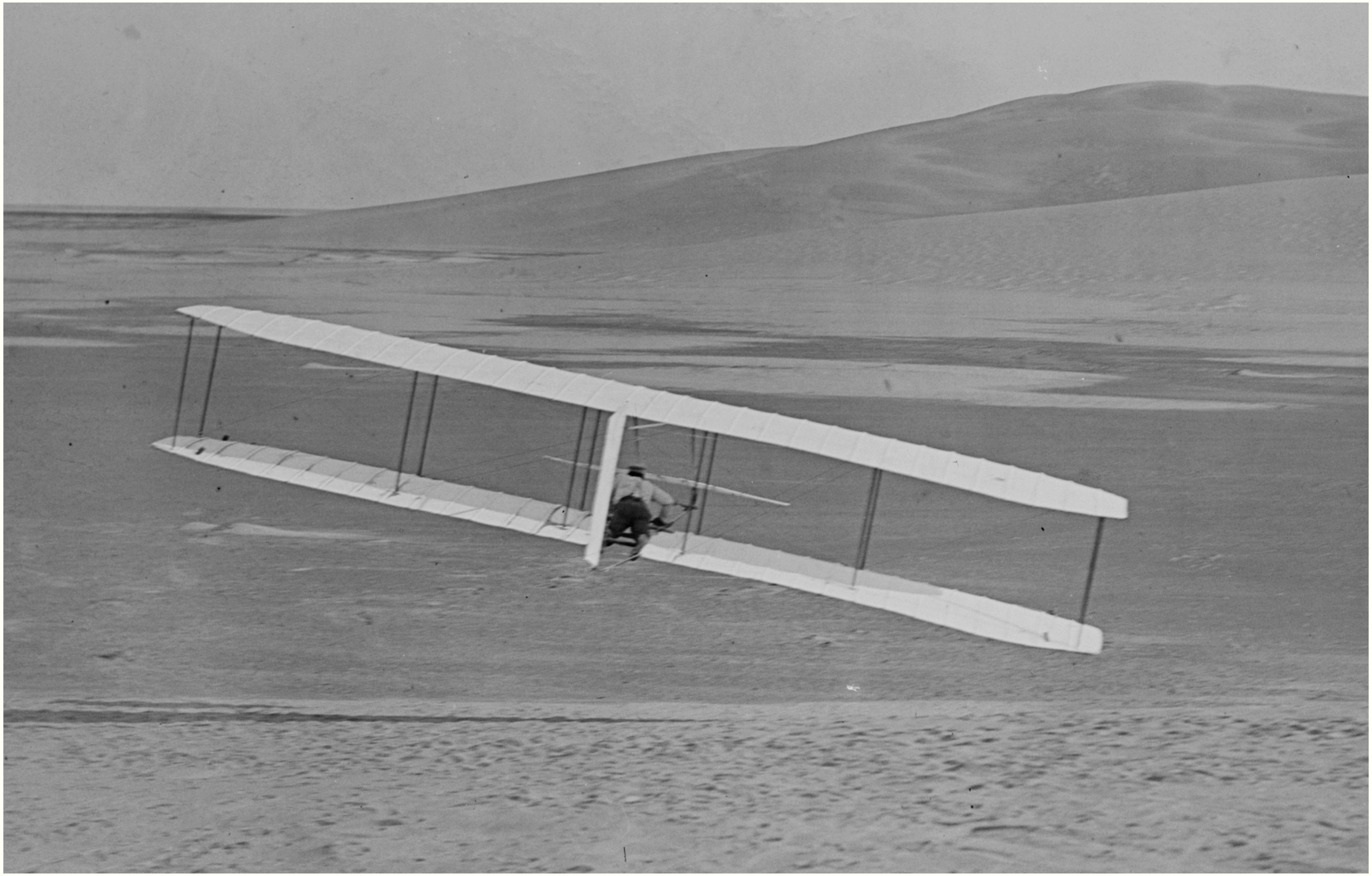
Invented and built the world's first powered airplane

Pictures from http://en.wikipedia.org/wiki/Wright_brothers

Crumpled gilder, Oct 1900

http://www.theatlantic.com/photo/2014/08/first-flight-with-the-wright-brothers/100796/

Glider (i.e. no power) (1902)



First powered flight (17 Dec 1903)
(Added: Propeller, engine)

Classical engineering design iteration

1. Design
   – This step may use calculations, physical laws, chemistry or biology, experimental data, intuition and guesses
2. Build
3. Test
4. If it doesn't work, go back to design (Step 1).

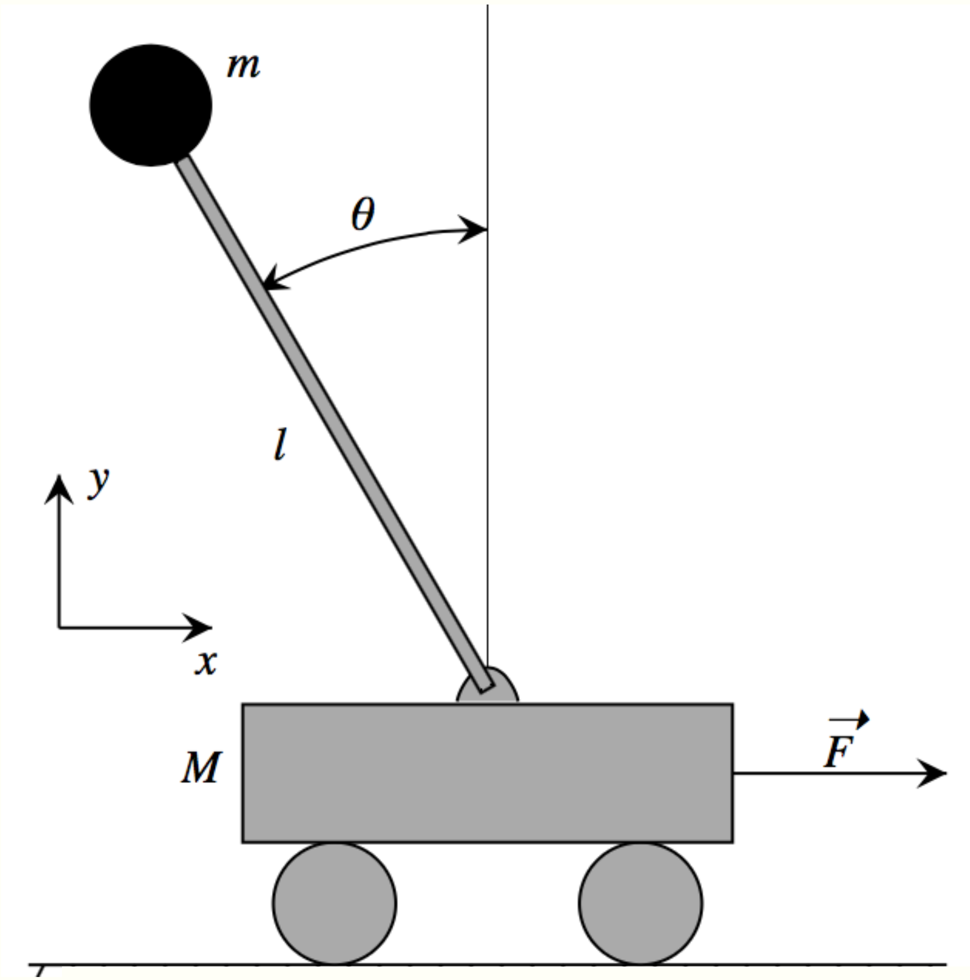
Engineering design iteration – with computers

1. Design on computers
   a) Derive **mathematical model** of the design
   b) Perform calculations, **simulations** or optimisation to understand or improve design
   c) Reject designs with poor performance. If none of the designs is good, go back to (a) for a new design or (b) to try to optimise the design.
   d) Choose one or more candidates for prototyping or building the actual design
2. Build
3. Test
4. If it doesn't work, go back to design (Step 1).

Mathematical model can be derived from science (maths, physics, chemistry, biophysics) or data

Design challenge: Balancing an inverted pendulum

- Can you balance a stick on your finger tip/palm

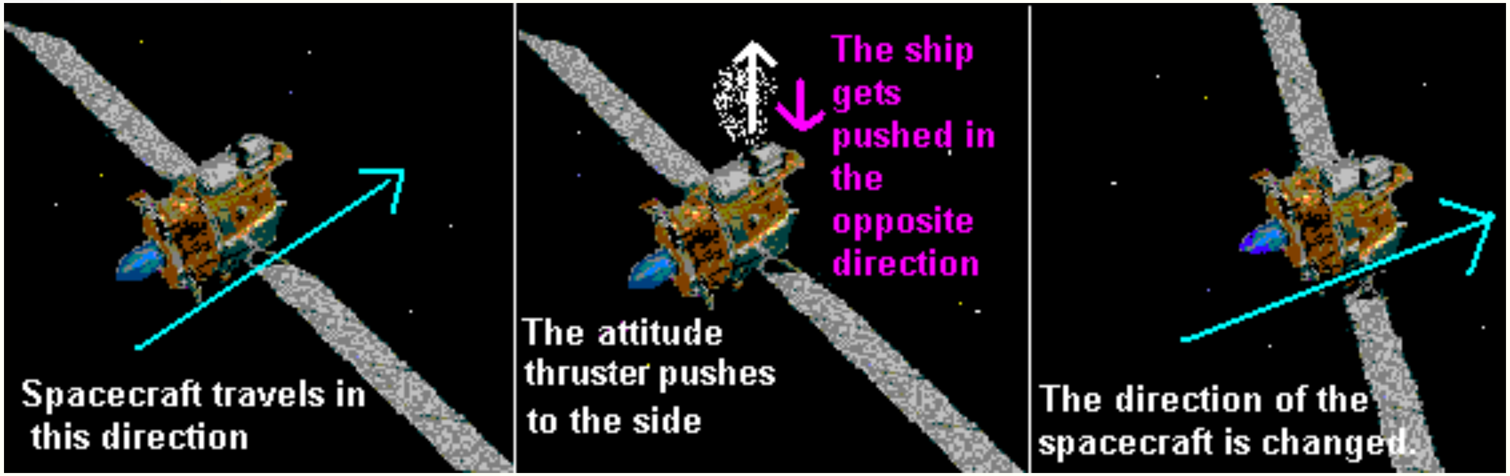


- An inverted pendulum is sitting on a cart.
- The aim of the design is to balance the inverted pendulum by applying an appropriate force on the cart.

Picture: http://en.wikipedia.org/wiki/Inverted_pendulum

Applications of inverted pendulum

- Segway
- Rocket/spaceship attitude control
  - i.e. orientation control



Picture http://www.segway.com/

http://www.qrg.northwestern.edu/projects/vss/docs/propulsion/2-what-is-attitude-control.html

This week

- Simulation

- Python components
  - Some new numpy functions

- Mathematical / physics / chemistry concepts
  - Mathematical modelling
  - Numerical approximation of derivatives
  - Ordinary differential equations

More on numpy

- Before looking at simulation, we will first go through a number of numpy functions which are related to our discussion this week

- The file is in numpy_ex.py

Notation in the lecture notes

- We will be using both mathematical variables and Python variables in this lecture

- We may say the position of an object at time t is $x(t)$
  - For example, x(0.3) = 5 says that the object is at the position 5 at time 0.3

- We may store the position of the object in a numpy array

Notation

Real number with interpretation of time

- Mathematical variable : x(0.4)

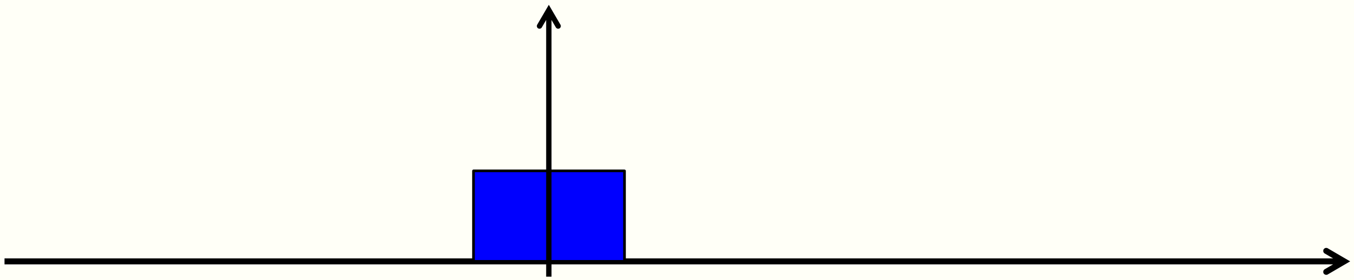
- Numpy array: position[5]

Zero or positive integers only
An index to an array

- A simple way to remember:
  - Mathematical variables: ()
  - Numpy array: []

Simulation on paper – the setup

- An object is constrained to move along a straight line
- Time starts at 0 unit. The initial position of the object is $x(0) = 1$
- The velocity $v(t)$ at time t is:
  - $v(t) = 2$ if $0 \le t < 0.4$
  - $v(t) = -4$ if $0.4 \le t < 0.8$
  - $v(t) = 1$ if $0.8 \le t$
- Determine the position of the object at $t = 0.1, 0.2, \ldots, 1$



At position 1 at time 0

Calculating positions on paper

- Given:
  - Initial position x(0) = 1
  - Velocity in time interval [0,0.1] is 2
- Aim: Find the position at time 0.1 = x(0.1)

- $x(0.1) = x(0) + 2 * 0.1 = 1.2$

- How about position at time 0.2 = x(0.2)
  - Velocity in time interval [0.1,0.2] is 2

- $x(0.2) = x(0.1) + 2 * 0.1 = 1.4$

Quiz: Position at time 0.3

- Given
  - $x(0.2) = 1.4$
  - Velocity in time interval [0.2,0.3] is 2
- What is the position at time 0.3?
  - Equivalently: What is $x(0.3)$?

Python variable for time instances

- Our aim is to compute the position of the object at time instances 0, 0.1, 0.2, … , 1

- We want to create a numpy array whose elements are:
  - [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
  - Python variable name for this numpy array: time_array
  - We can generate this array by using either arange() or linspace()

Python variable for positions

- time_array = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]

- pos_array = [1, 1.2, 1.4, …]

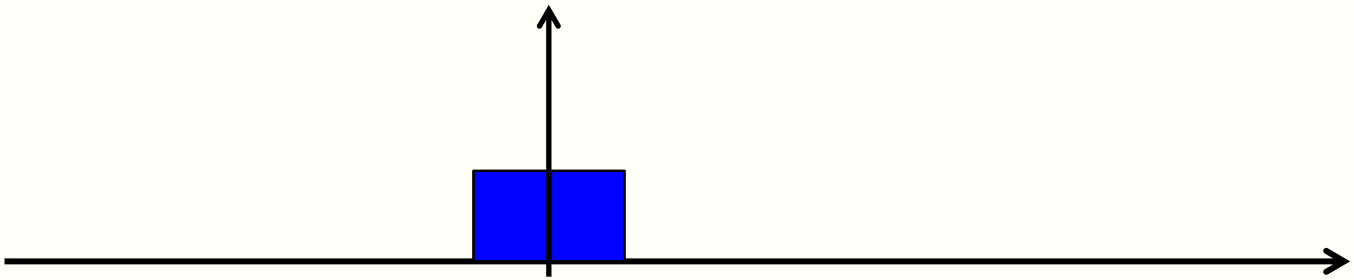
- pos_array and time_array have the same shape

- Note:
  - pos_array[0] = position at time 0 = position at time time_array[0]
  - pos_array[1] = position at time 0.1 = position at time time_array[1]

- Generally:
  - pos_array[k] = position at time 0.1*k = position at time time_array[k]

Simulation on paper – the setup (repeat)

- An object is constrained to move along a straight line
- Time starts at 0 unit. The initial position of the object is $x(0) = 1$
- The velocity $v(t)$ at time t is:
  - $v(t) = 2$ if $0 \leq t < 0.4$
  - $v(t) = -4$ if $0.4 \leq t < 0.8$
  - $v(t) = 1$ if $0.8 \leq t$
- Determine the position of the object for t = 0.1,0.2, …, 1



At position 1 at time 0

Simulation on paper

Index k	time_array[k]	Speed in the interval before t	Position pos_array[k]
0	0.0		pos_array[0] = 1
1	0.1	2	pos_array[1] = pos_array[0] + 2 * 0.1 = 1.2
2	0.2	2	pos_array[2] = pos_array[1] + 2 * 0.1 = 1.4
3	0.3	2	pos_array[3] = pos_array[2] + 2 * 0.1 = 1.6
4	0.4	2	pos_array[4] = pos_array[3] + 2 * 0.1 = 1.8
5	0.5	-4	pos_array[5] = pos_array[4] - 4 * 0.1 = 1.4
6	0.6	-4	pos_array[6] = pos_array[5] - 4 * 0.1 = 1.0
7	0.7	-4	pos_array[7] = pos_array[6] - 4 * 0.1 = 0.6
8	0.8	-4	pos_array[8] = pos_array[7] - 4 * 0.1 = 0.2
9	0.9	1	pos_array[9] = pos_array[8] + 1 * 0.1 = 0.3
10	1.0	1	pos_array[10] = pos_array[9] + 1 * 0.1 = 0.4

Let us complete the Python implementation in simulate_1d_prelim.m

simulate_1d.py (simulation loop only)

```
for k in range(1,len(time_array)):
    #  Find the velocity in the previous time interval
    #  Find the previous time instance
    time_before = time_array[k-1]
    if time_before < TIME_LIMIT_1:
        velocity = VELOCITY_1
    elif time_before < TIME_LIMIT_2:
        velocity = VELOCITY_2
    else:
        velocity = VELOCITY_3

    #  Update pos_array(k)
    pos_array[k] = pos_array[k-1] + velocity * dt
```

Quiz: How can we improve simulate_1d.py?

```
for k in range(1,len(time_array)):
    #  Find the velocity in the previous time interval
    #  Find the previous time instance
    time_before = time_array[k-1]
    if time_before < TIME_LIMIT_1:
        velocity = VELOCITY_1
    elif time_before < TIME_LIMIT_2:
        velocity = VELOCITY_2
    else:
        velocity = VELOCITY_3

    #  Update pos_array(k)
    pos_array[k] = pos_array[k-1] + velocity * dt
```

Quiz: How can you improve this code?

Hint: You may want to move this this section of code to a ???

Week 3's lecture (1)

- Speed of an object in freefall

$$v(t) = \frac{gm}{d}\left(1 - e^{-\frac{d}{m}t}\right)$$

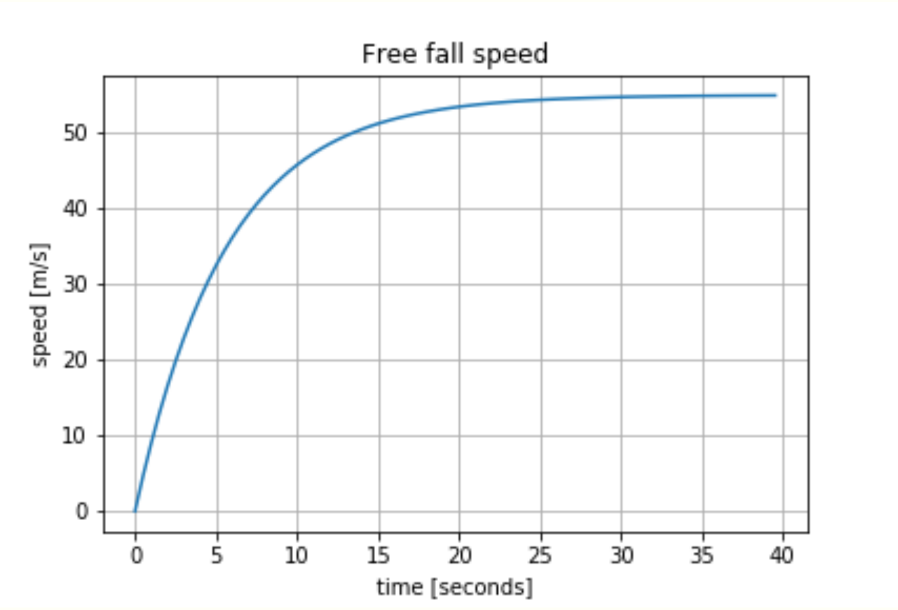
- You created a list of time instances

[0, 0.5, 1, 1.5, 2, 2.5, 39.5, 40]

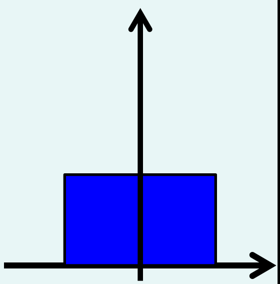
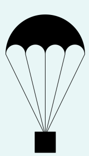
In lecture project in Week 3 (2)

- You use for-loops to create a list of speeds
  - Time is 0. Use the speed formula. Speed = 0.
  - Time is 0.5. Use the speed formula. Speed = 4.692400935
  - Time is 1. Use the speed formula. Speed = 8.98399681455

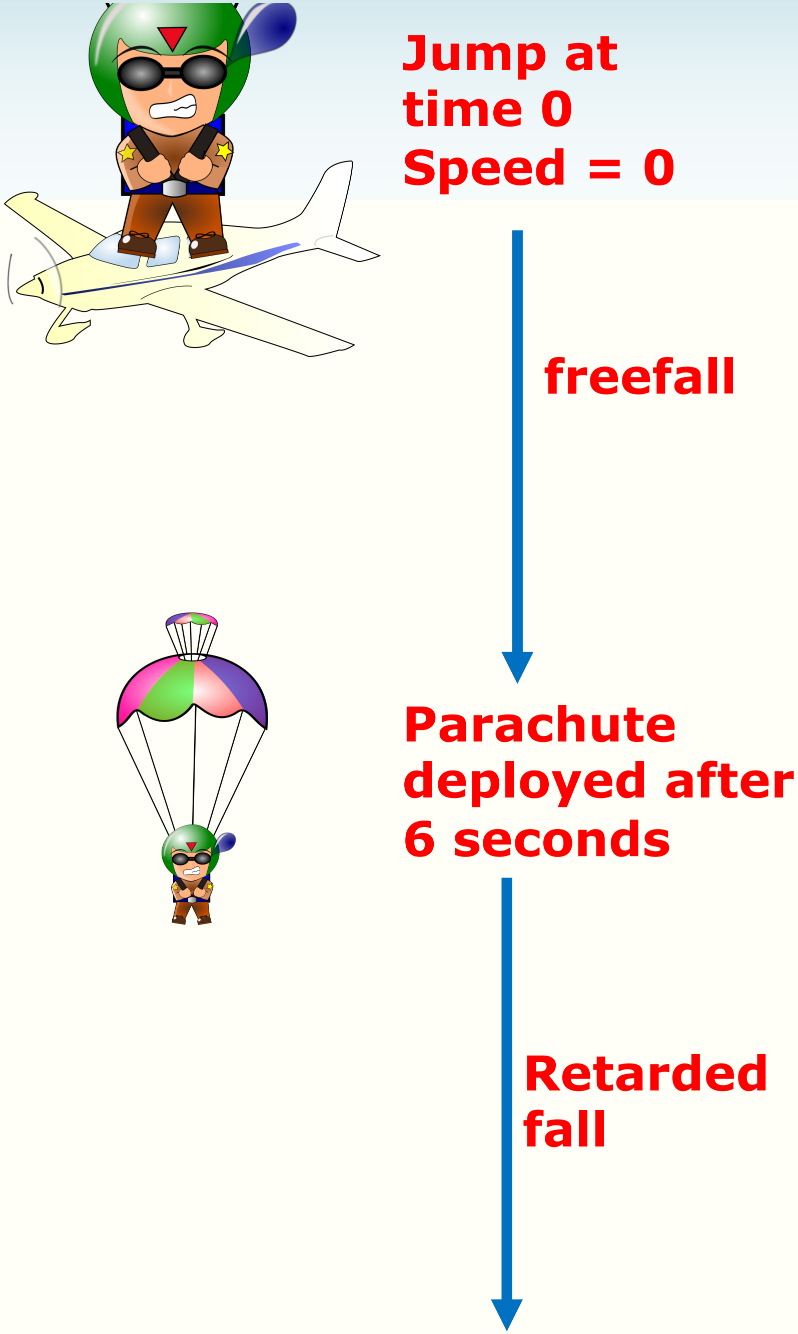
  - Time is 40. Use the speed formula. Speed = 54.8885179036



Contrasting two methods to do simulation

	By increment	By Formula
	pos_array[k] = pos_array[k-1] + velocity * dt	
		$v(t) = \frac{gm}{d}\left(1 - e^{-\frac{d}{m}t}\right)$

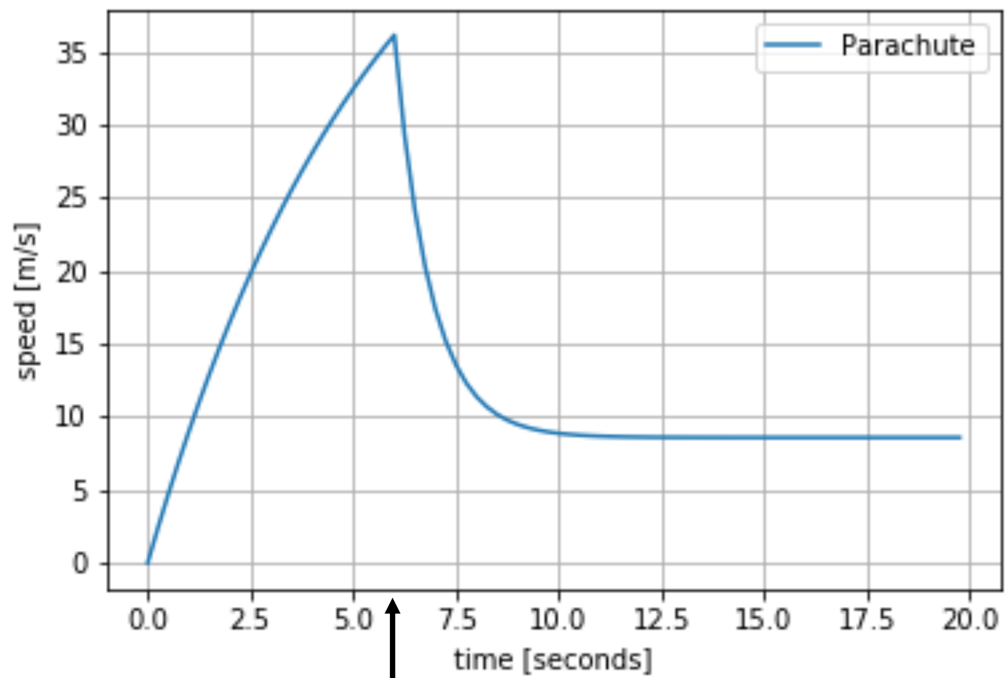
Simulation by formula



- A parachutist jumps from the plane, we want to calculate their speed over time and plot the speed profile

The final product

- We will need two formulas
  - One before the parachute is deployed: freefall
  - One after the parachute is deployed



Time at which the parachute is deployed

Before the parachute is deployed

- Notation:
  - m is the mass of the parachutist
  - g is acceleration due to gravity ($\mathrm{m\,s^{-2}}$)
  - c_{air} is the drag coefficient in air (in $\mathrm{kg\,s^{-1}}$)
  - t_c is the time the parachute is deployed
- The velocity of the parachutist before the parachute is deployed is given by the formula:

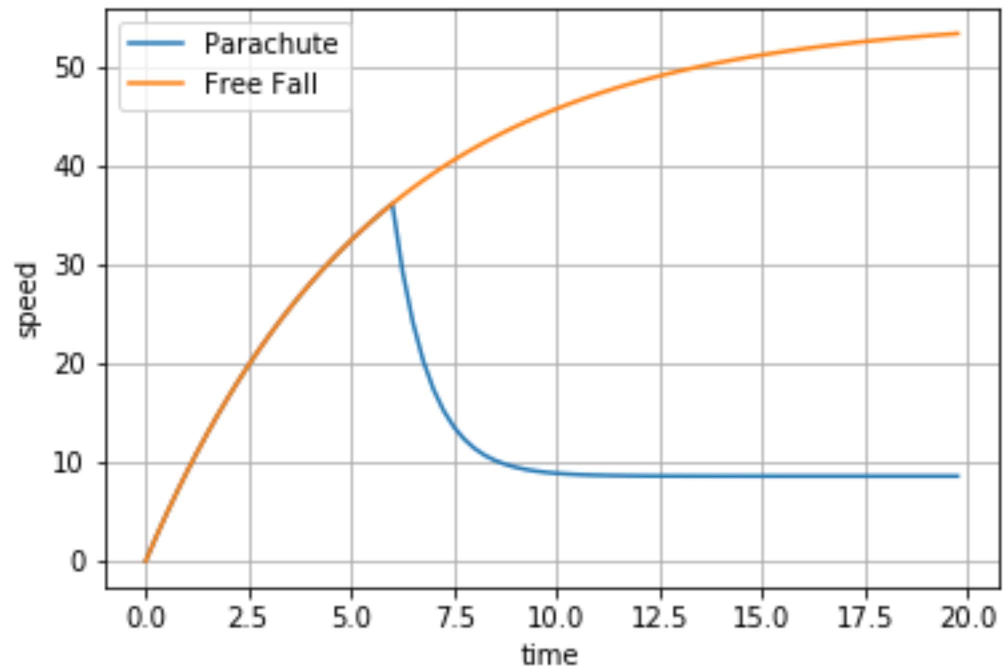
Free fall part

If $t < t_c$

$$v(t) = \frac{gm}{c_{\mathrm{air}}}\left(1 - e^{-\frac{c_{\mathrm{air}}}{m}t}\right)$$

Some intuition

$$v(t) = \frac{gm}{c_{\text{air}}}\left(1 - e^{-\frac{c_{\text{air}}}{m}t}\right)$$



The exponential factor decays in magnitude, so the speed asymptotically approaches $g\,m/\,c_{air}$

For a free-falling 70kg parachutist with c_{air} = 12.5, this **terminal speed** is ~55 m s^{-2} (200km/hr)

After the parachute is deployed

- t_c is the time at which the parachute is deployed
- A larger drag coefficient c_{dp}

If $t \geq t_c$

$$v_{p0} = \frac{gm}{c_{\text{air}}}\left(1 - e^{-\frac{c_{\text{air}}}{m} t_c}\right)$$

Speed at the moment parachute is deployed

$$v(t) = v_{p0} e^{-\frac{c_{dp}}{m}(t - t_c)} + \frac{gm}{c_{dp}}\left(1 - e^{-\frac{c_{dp}}{m}(t - t_c)}\right)$$

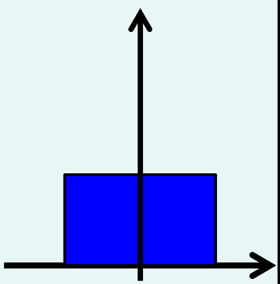
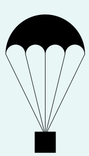
Parachutist simulation

- We can write a function that, given all parameters, calculates the velocity at any time `t`
- The algorithm, expressed in *pseudocode*, is

```
for t in time_array
    if t < tc    # still in free-fall
        Calculate the freefall velocity formula
    else
        Calculate velocity at time of deployment
        Calculate velocity parachute velocity formula
```

Code in para_speed_by_formula.py and para_formula_lib.py

Comparing object moving in 1D and parachutist

	By increment	By Formula
	**pos_array[k] = pos_array[k-1] + velocity * dt**	• **If $0 \le t \le 0.4$, $x(t) = 1+2t$** • **If $0.4 \le t \le 0.8$, $x(t) = 1.8 - t$** • **If $t \ge 0.8$, $x(t) = 0.2 + t$**
	?	$v(t) = \frac{gm}{d}\left(1 - e^{-\frac{d}{m}t}\right)$ **plus others.**

An inconvenient truth

- Solving problems by deriving a formula
  - Mathematically elegant; exact solution
  - Formulas may provide insight
  - Convenient to use: simply perform substitution

Bad news!

Most advanced engineering problems do not have an **exact** solution in the form of a formula

Good news!

You can solve these problems **numerically** and **approximately** by **computers** and **programming**

Non-formula solution to the parachutist problem

- The velocity of the parachutist obeys the following ordinary differential equations (ODE)

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

- v(t) = velocity at time t
- c(t) = drag coefficient at time t

- We will look at how you can solve this equation numerically and approximately.

Approximating derivatives

- From the definition of derivatives, we know

$$\frac{dv(t)}{dt} = \lim_{\Delta \to 0} \frac{v(t+\Delta) - v(t)}{\Delta}$$

- If Δ is small enough, then

$$\frac{dv(t)}{dt} \approx \frac{v(t+\Delta) - v(t)}{\Delta}$$

Approximating derivatives – numerical illustration

- $f(x) = x^3$
- Derivative of $f(x) = f'(x) = 3x^2$
- At $x = 2$, $f'(2) = 12$
- Let us compute the approximate derivative for different values of Δ

$$\frac{(2+\Delta)^3 - 2^3}{\Delta}$$

Code: approximate_derivative.py

Solving ODE numerically (1)

1) Starting from the ODE
$$\frac{dv(t)}{dt} = g - \frac{c(t)}{m} v(t)$$

2) Replace the derivative by its approximation
$$\frac{dv(t)}{dt} \approx \frac{v(t+\Delta) - v(t)}{\Delta}$$

We obtain:
$$\frac{v(t+\Delta) - v(t)}{\Delta} \approx g - \frac{c(t)}{m} v(t)$$

Solving ODE numerically (2)

Previous step:

$$\frac{v(t+\Delta) - v(t)}{\Delta} \approx g - \frac{c(t)}{m} v(t)$$

3) Make v(t + Δ) the subject:

$$v(t+\Delta) \approx v(t) + (g - \frac{c(t)}{m} v(t))\Delta$$

Solving ODE numerically (3)

Previous step:

$$v(t+\Delta) \approx v(t) + (g - \frac{c(t)}{m} v(t))\Delta$$

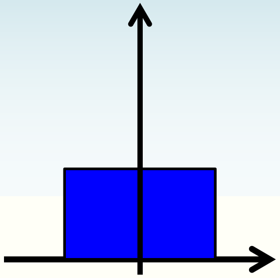
New speed **Old speed**

- For simulation, let us assume speed is stored in the array speed_array
- Identify

  v(t+Δ) with speed_array[k]

  v(t) with speed_array[k-1]

The analogy …



$$x(0.3) = x(0.2) + 2 * 0.1 = 1.6$$

Position after 0.1 time units

Position at time 0.2

Δ

```
pos_array[4] = pos_array[3] + 2 * 0.1 = 1.6
```

Next element in the vector

Previous element in the vector

Python code: approx ODE versus formula

- A Python function to solve the ODE numerically for the parachutist problem
  - Solution in the function: para_ODE_lib.py

- Note
  - Formula is exact
  - Numerical solution to ODE is an **approximation**

- Python script para_speed_by_ODE.py compares the formula against the approximate numerical solution

- We will vary the value of Δ, we expect
  - Small Δ, small difference between the two methods
  - And vice versa

Where did the ODE come from?

ODE we used. Multiply both sides by m.

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

Let us look at what this means.

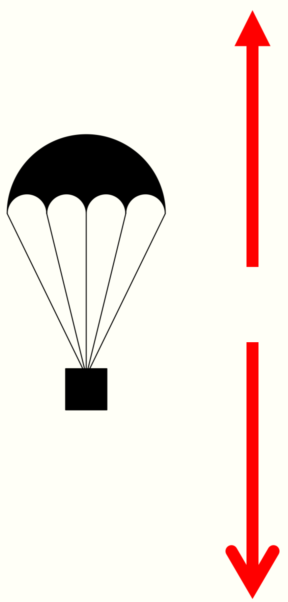
$$m\frac{dv(t)}{dt} = mg - c(t)v(t)$$

ODEs describe physical laws

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

mass x acceleration = Net downward force on the parachutist

Newton's second law



c(t) v(t) = drag force

m g = gravitational pull

The big picture

- Physical law gives the ODE

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

- Computers and algorithms allow you to obtain numerical and approximate solution

- That's why you need to learn maths, physics, chemistry, your own disciplinary knowledge and COMPUTING!

Solving ODEs

- The method we use for solving ODE is known as **Euler's forward method**
- Meaning of forward and backward:

Forward: $$\frac{dv(t)}{dt} \approx \frac{v(t+\Delta) - v(t)}{\Delta}$$

Backward: $$\frac{dv(t)}{dt} \approx \frac{v(t) - v(t-\Delta)}{\Delta}$$

- Euler's forward method is simpler to explain but **not** the best. This is so you can focus on learning programming
- You will learn better methods in later years

The extended parachutist problem

- What if you want to determine the height of the parachutist too?
- Let h(t) = height of the parachutist at time t
- How can you compute h(t + Δ) from h(t)?

$$h(t + \Delta) \approx h(t) - v(t)\Delta$$

New height **Old height**

- You can formally derive this from the following ODE which says: derivative of height = downward speed

$$\frac{dh(t)}{dt} = -v(t)$$

Python implementation

- Essentially, two updates in the for loop

$$v(t+\Delta) \approx v(t) + (g - \frac{c(t)}{m} v(t))\Delta$$

$$h(t+\Delta) \approx h(t) - v(t)\Delta$$

- Python function: para_ODE_ext_lib.py
- Python script: para_speed_height_by_ODE.py
  - The script also illustrates how to plot with two different scales for the y-axis

para_ODE_ext_lib.py

```
def para_speed_height_ODE(time_array, mass, speed0,
               height0, drag_air, time_deploy, drag_para):

    height_array = np.zeros_like(time_array)

    height_array[0] = height0

    # simulation loop
    for k in range(1,len(time_array)):

        height_array[k] = height_array[k-1] - \
                          speed_array[k-1] * dt
```

Summary

- We have introduced the basics of simulation, which is a key tool in modern engineering and science
  - A formula solution is rare for modern day complex engineering problems
  - Numerical solution, approximation solution and simulation are important methods
- The basic method to do simulation is to set up an iteration step which can be obtained from ordinary differential equations