

B1 Numerical Algorithms

4 Lectures, MT 2023

Lecture one – Computing derivatives and integrals, errors and convergence.

Wes Armour

23rd October 2023

Objective of this lecture

Today we will cover the basics of numerical analysis and methods. These basic concepts underpin much of computational methods you will use as Engineers.

We will use what we learn today to build on in our forthcoming lectures to understand topics such as...

Working with PDEs



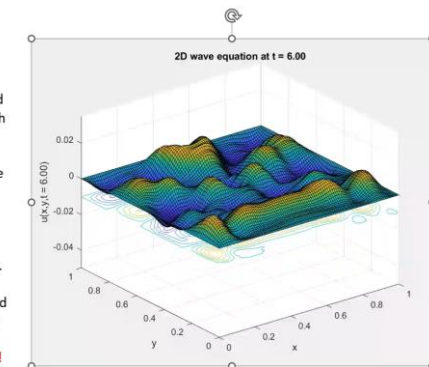
A vapour cone is a visible cloud of condensed water that can form around an object moving at high speed through moist air.

When the local air pressure around the object drops, so does the local air temperature.

If the temperature drops below the saturation temperature, a cloud forms.

This phenomena typically occurs around aircraft travelling at transonic speeds

So, very complicated physics to model!



Second order equations

20 Steps

- Euler all over the place
- Predictor-corrector begins to approximate the function well.

40 steps

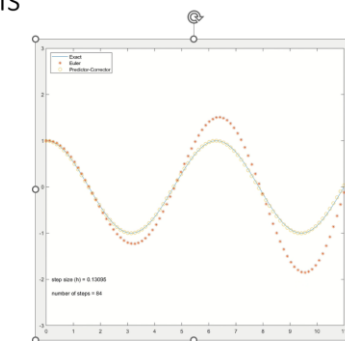
- Euler oscillating wildly
- Predictor-corrector getting close to real function.

60 steps

- Euler still far from the actual function
- Predictor-corrector very close to real function.

80 steps

- Game over – predictor-corrector is almost exact!



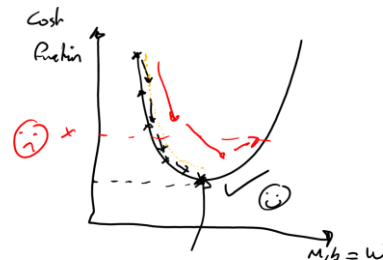
Machine learning – Learning Rate

We need to step along our cost function curve in the direction of the negative gradient.

The size of our steps are called the **learning rate**.

A large step size (learning rate) could cause us to step over the minima, missing it.

A small learning rate though, means many tiny steps, taking a very long time to complete.

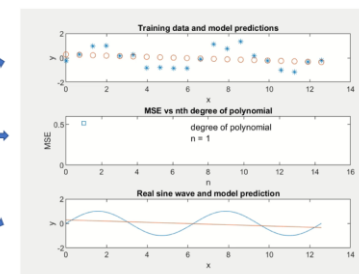


Machine Learning and Overfitting

This example fits a polynomial of degree M to a noisy sine wave.

Look at the Mean Squared Error as the polynomial degree changes.

Finally let's take a look at how our polynomial model actually fits our data:



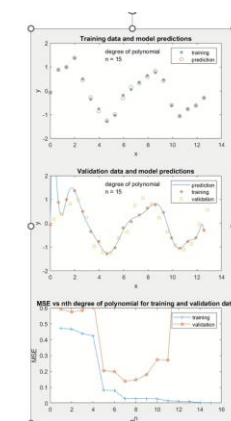
Validating our model

How do we know when we have fitted our model (our polynomial) correctly? Or put another way, how do we evaluate our model's skill at prediction?

We use validation data!

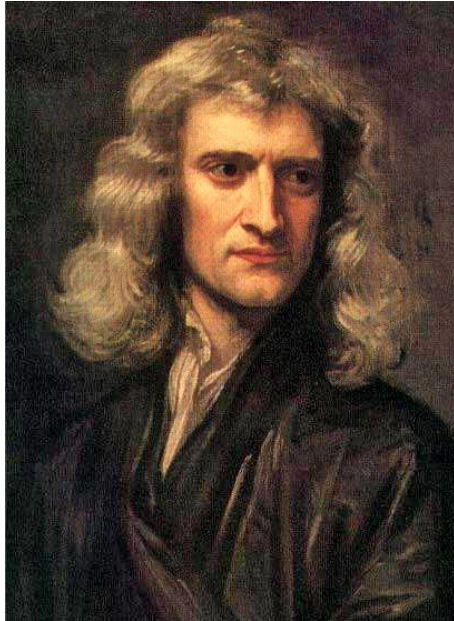
Here you can see how I've held back data from our model training (yellow squares), and used this to test the predictive skill of our model (MSE, third plot, stars). We can see when $n \sim 7$ our model has the lowest MSE for the training data and the validation data that it hasn't seen before.

One last thing, in this context, n could be viewed as a **hyperparameter** of our model.



Learning outcomes

- Part A - A toy experiment
- Part A - Sources of error and convergence
- Part B - Finite difference approximations (*numerical differentiation...*)
- Part B - Convergence
- Part C - Numerical quadrature (*also known as numerical integration...*)



Isaac Newton
1642 – 1727



Gottfried Leibniz
1646 - 1716

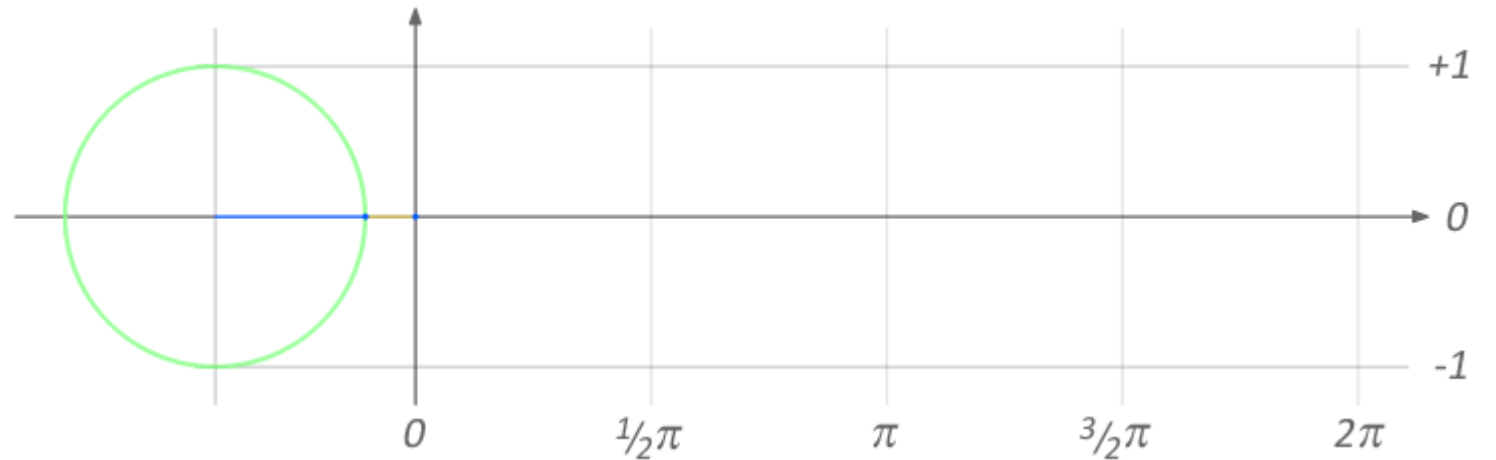


Brook Taylor
1685 - 1731

A position sensor



Imagine we have a position sensor, that measures the height of Harry (*his altitude*) during his ride on a Ferris wheel.



How to do this...

```
BMP085test | Arduino 1.8.9 (Windows Store 1.8.21.0)
File Edit Sketch Tools Help

BMP085test $
#include <Wire.h>
#include <Adafruit_BMP085.h>

/*****
 * This is an example for the BMP085 Barometric Pressure & Temp Sensor
 *
 * Designed specifically to work with the Adafruit BMP085 Breakout
 * ----> http://www.adafruit.com/products/391
 *
 * These displays use I2C to communicate, 2 pins are required to
 * interface
 * Adafruit invests time and resources providing this open source code,
 * please support Adafruit and open-source hardware by purchasing
 * products from Adafruit!
 *
 * Written by Limor Fried/Ladyada for Adafruit Industries.
 * BSD license, all text above must be included in any redistribution
 *****/

// Connect VCC of the BMP085 sensor to 3.3V (NOT 5.0V!)
// Connect GND to Ground
// Connect SCL to I2C clock - on '168/'328 Arduino Uno/Duemilanove/etc thats Analog 5
// Connect SDA to I2C data - on '168/'328 Arduino Uno/Duemilanove/etc thats Analog 4
// EOC is not used, it signifies an end of conversation
// XCLR is a reset pin, also not used here

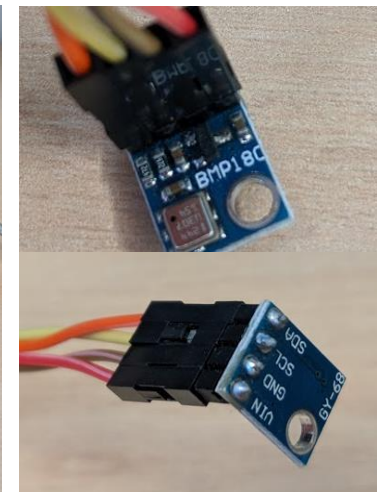
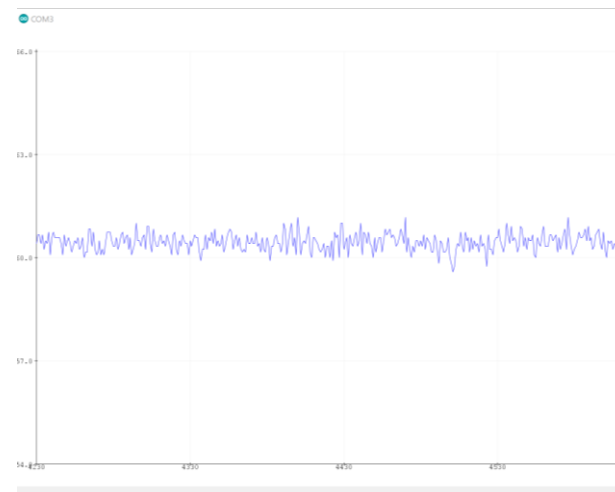
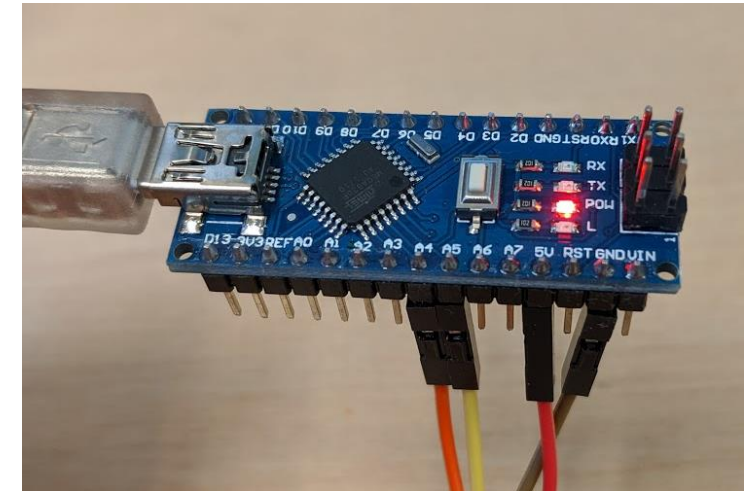
Adafruit_BMP085 bmp;

void setup() {
  Serial.begin(9600);
  if (!bmp.begin()) {
    Serial.println("Could not find a valid BMP085 sensor, check wiring!");
    while (1) {}
  }
}

void loop() {
  // Set the bmp.readAltitude() argument (below it's 102100) to the barometric pressure at
  // sea level for the time of your measurement - use http://www.bbc.co.uk/weather/2635650
  Serial.print(bmp.readAltitude(102100));
  Serial.println();
  delay(500);
}

Sketch uses 7406 bytes (24%) of program storage space. Maximum is 30720 bytes.
Global variables use 485 bytes (23%) of dynamic memory, leaving 1563 bytes for local variables. Maximum is 2048 bytes.
```

1. Arduino Nano (right)
2. BMP280 barometric pressure sensor (bottom right)
3. Adafruit code (left)
4. Wiring (left pictures)
5. Output in meters (below)



Our measurement

Imagine our BMP180 sensor measures Harrys altitude every 5 seconds.

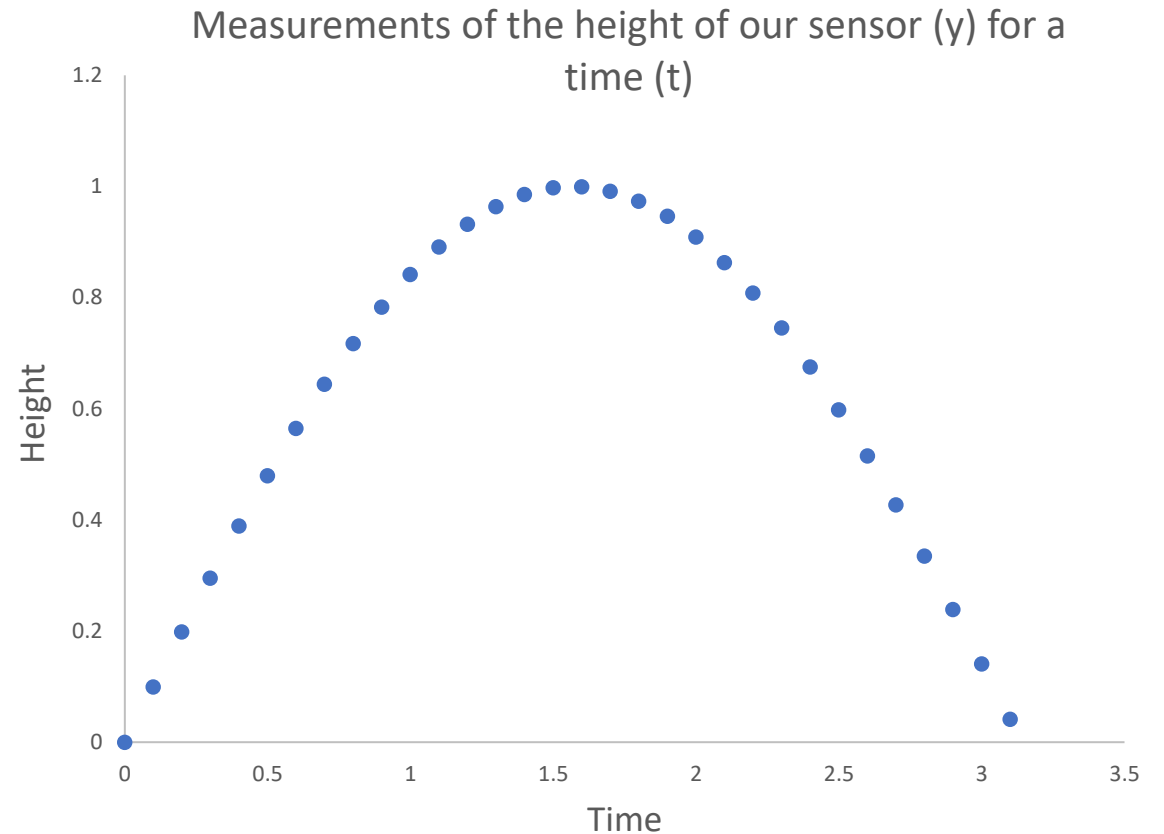
It produces the following data:

$$\underline{D} = (t_1, y_1), (t_2, y_2) \dots (t_N, y_N)$$

Where y is altitude at time point t .

The physics of the system means that the data is described by a sine function
(see previous animation).

$$f(t) = \sin(t)$$



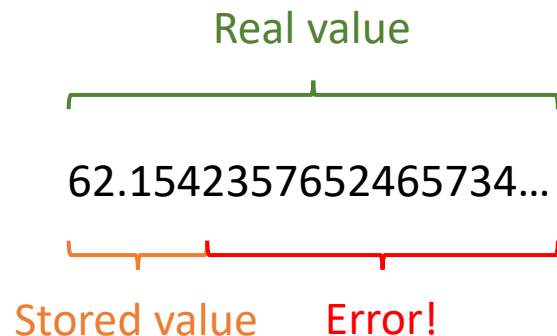
Sources of errors in our measurements

We will use a barometer sensor (measuring pressure) to estimate altitude (predict height).

Where are the sources of error in doing this?

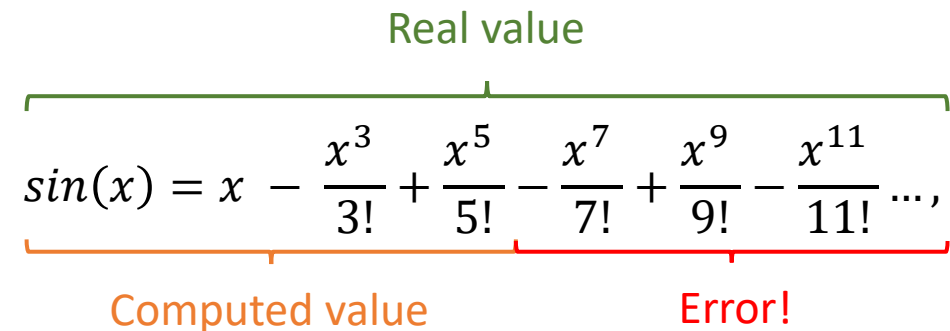
Round-off error

A pressure measurement is represented by a finite number. These finite numbers will then be divided, subtracted... Each time we lose some precision due to the finite approximation of the real value.



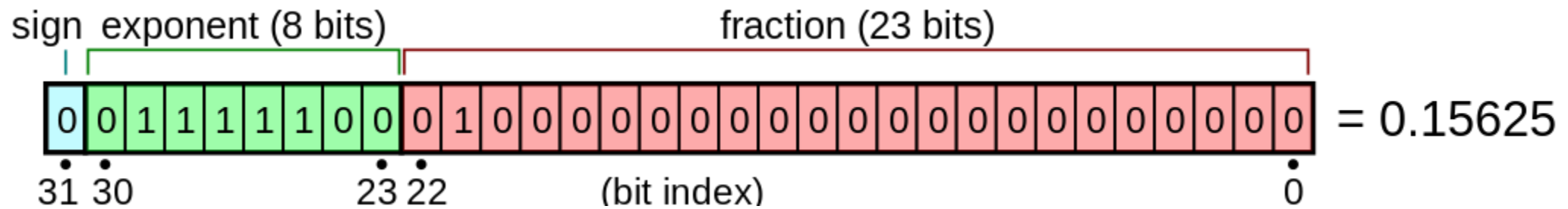
Truncation error

The error caused by representing an infinite sum with a finite one. Our Arduino can't keep computing infinitely, we would never get an answer!



Floating points

A *single precision* floating point number is represented by 32 bits of information. It looks like:



- Sign bit: 1 bit
- Exponent width: 8 bits
- Significand precision: 23 bits

IEEE 754 has different versions of “floats”:

Half precision (fp16) uses 16 bits

Single precision (fp32) uses 32 bits

Double precision (fp64) uses 64 bits

Gapping



What is the number “next to” 0.15625? - To find out we flip the lowest bit

$$= 0.156250014901161193848$$


**So any number in between 0.15625 and 0.156250014901161193848 cannot be represented exactly
it needs to be rounded to the nearest number**

Numerical representations and details

Panopto bonus video

B1 Numerical Algorithms

4 Lectures, MT 2022
Lecture one
Bonus video – Rounding errors and storing numbers digitally
Wes Armour



Errors/Typos/Questions in wa@armour.cba.uq.edu.au

Bits, Bytes and nibbles

Before we think about storing continuous numbers digitally, let's look at how an integer number is stored.

Current computer hardware uses the bit as a mechanism to store numbers. It can take on two values, 0 or 1.

A **byte** is a collection of 8 bits.

4 bytes (32 bits) are combined to form an **int**.

8 bytes (64 bits) are combined to form a **long int**.

To understand how this combining works, let's take a look at a **nibble** – a 4 bit integer.

Note that computer programming languages and hardware sometimes employ different representations to those above. See here: [https://en.wikipedia.org/wiki/Integer_\(computer_science\)#Common_integer_data_types](https://en.wikipedia.org/wiki/Integer_(computer_science)#Common_integer_data_types)

The nibble

A nibble is a combination of 4 bits. Each bit can be either 0 or 1.

So how many different things can a nibble represent?

1 bit = 1 thing $2^1 = 2$
2 bits = 4 things $2^2 = 4$
3 bits = 8 things $2^3 = 8$
4 bits = 16 things $2^4 = 16$

How do we use these combinations to represent integer numbers?

$$5_{10} = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 0 + 1 = 5$$

$$146_{10} = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 128 + 64 + 32 + 16 + 0 + 4 + 0 + 1 = 241$$

Storing continuous numbers digitally

All mainstream modern computing hardware implement the IEEE Standard for Floating-Point Arithmetic: [IEEE 754](https://en.wikipedia.org/wiki/IEEE_754).

This standard ensures that everyone agrees on how to store a continuous number, how to perform arithmetic operations, how to round numbers (for example when adding them together), along with other things.


A floating point number is an approximate representation of a continuous number.

The term floating point reflects the fact that the **decimal point** in the representation can "float". It can be placed anywhere within the significant digits of the number.

Let's look at this in some more detail.

Floating points

A single precision floating point number is represented by 32 bits of information. It looks like:



sign, exponent (8 bits), fraction (23 bits) = 0.15625

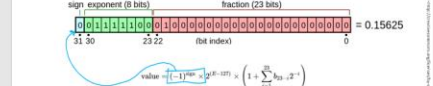
Bit indices: 31, 30, 23, 22, 0

Sign bit: 1 bit
Exponent width: 8 bits
Significant precision: 23 bits

IEEE 754 has different versions of "floats":
Half precision (fp16) uses 16 bits
Single precision (fp32) uses 32 bits
Double precision (fp64) uses 64 bits

Floating points – example

Let's think about our example below – how do we arrive at the number 0.15625?



sign, exponent (8 bits), fraction (23 bits) = 0.15625

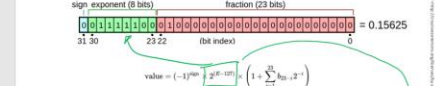
Bit indices: 31, 30, 23, 22, 0

value = $(-1)^{\text{sign}} \times 2^{(\text{exponent}-127)} \times (1 + \sum_{n=1}^{23} b_n \times 2^{-n})$

$(-1)^0 = 1$

Floating points – example

Let's think about our example below – how do we arrive at the number 0.15625?



sign, exponent (8 bits), fraction (23 bits) = 0.15625

Bit indices: 31, 30, 23, 22, 0

value = $(-1)^{\text{sign}} \times 2^{(\text{exponent}-127)} \times (1 + \sum_{n=1}^{23} b_n \times 2^{-n})$

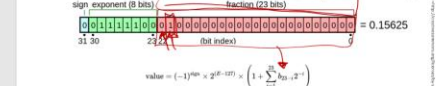
$E = 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 124$

$E = \sum_{n=0}^7 b_n \times 2^n = 124$

$124 \div 80 = 1.55 \approx 1.5625$

Floating points – example

Let's think about our example below – how do we arrive at the number 0.15625?



sign, exponent (8 bits), fraction (23 bits) = 0.15625

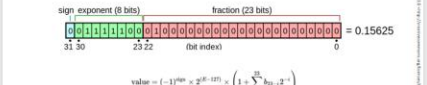
Bit indices: 31, 30, 23, 22, 0

value = $(-1)^{\text{sign}} \times 2^{(\text{exponent}-127)} \times (1 + \sum_{n=1}^{23} b_n \times 2^{-n})$

$124 \div 80 = 1.55 \approx 1.5625$

Floating points – example

Let's think about our example below – how do we arrive at the number 0.15625?



sign, exponent (8 bits), fraction (23 bits) = 0.15625

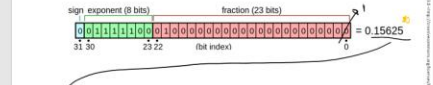
Bit indices: 31, 30, 23, 22, 0

value = $(-1)^{\text{sign}} \times 2^{(\text{exponent}-127)} \times (1 + \sum_{n=1}^{23} b_n \times 2^{-n})$

$124 \div 80 = 1.55 \approx 1.5625$

Gapping

What is the number "next to" 0.15625?



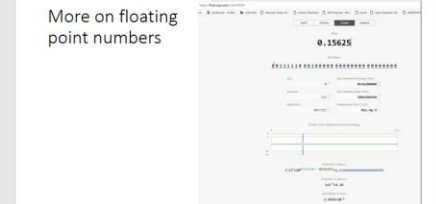
sign, exponent (8 bits), fraction (23 bits) = 0.15625

Bit indices: 31, 30, 23, 22, 0

value = $(-1)^{\text{sign}} \times 2^{(\text{exponent}-127)} \times (1 + \sum_{n=1}^{23} b_n \times 2^{-n})$

$124 \div 80 = 1.55 \approx 1.5625$

More on floating point numbers



What about letters, symbols...?

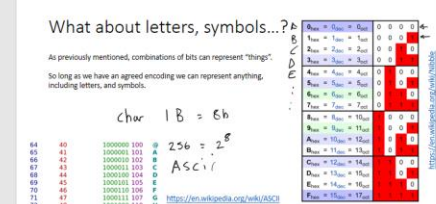
As previously mentioned, combinations of bits can represent "things".

So long as we have an agreed encoding we can represent anything, including letters, and symbols.

char 1 B = 8 b

256 = 2^8

ASCII



Part B – Finite difference
approximations

Estimating Derivatives.

Engineers often need to calculate derivatives approximately, either from data or from functions for which simple analytic forms of the derivatives don't exist.

Some examples of this:

- Motion simulation, such as in flight simulators solving $\ddot{x} = \vec{F}$ equations.
- Estimation of rates of change of measured signals.
- Control systems, where e.g. velocity signals are wanted from *position sensor data*.
- Medical signal & image processing – analysis and visualisation

Most methods derive from the basic derivation of differentiation of a function $f(t)$:

$$f' = \frac{df}{dt} = \lim_{\delta t \rightarrow 0} \frac{f(t + \delta t) - f(t)}{\delta t}$$

The rate of change of altitude

Suppose we wish to answer the question –

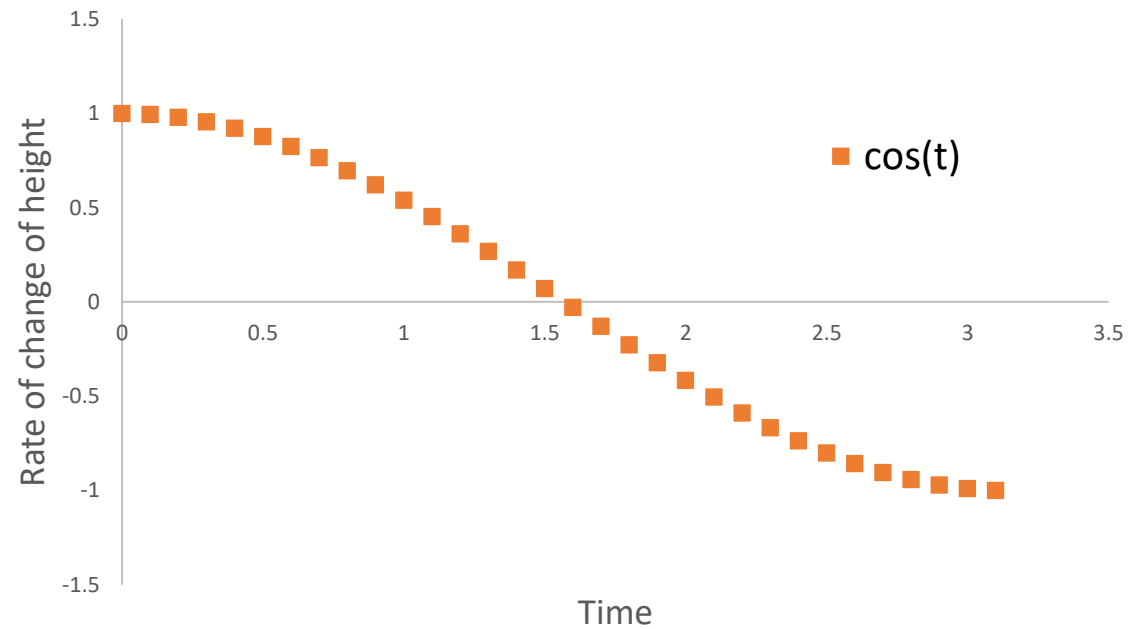
What is Harry's vertical velocity? i.e. his rate of change of altitude (y) with respect to time?

To answer this we turn to simple calculus:

$$\frac{df(t)}{dt} = f'(t) = \frac{d}{dt} \sin(t)$$

$$f'(t) = \cos(t)$$

Rate of change of altitude of our sensor (y) in time



Numerical differentiation

Now suppose **that we don't know** our sensor data is described by:

$$f(x) = \sin(t)$$

And its derivative by:

$$f'(x) = \cos(t)$$

Given the data, how would we find Harry's vertical velocity?

To do this, let's look at three basic schemes that answer this question:

- Forward difference
- Backward difference
- Central difference

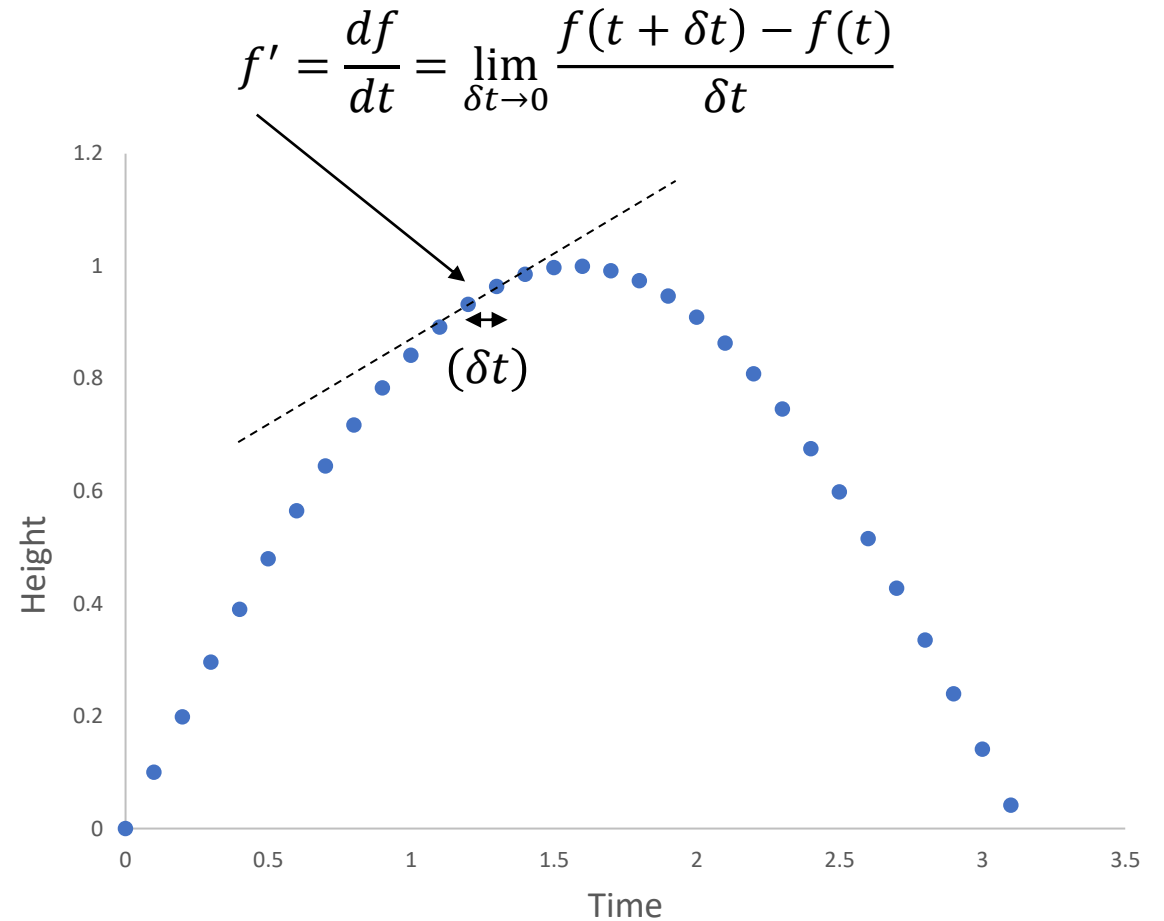
But first let's revisit our definition of a derivative.

Estimating Derivatives

Recall that the definition of a derivative is the gradient of the line that is tangent to our function.

To find this we look at the gradient between two points that are very close together.

In the limit of the second point becoming closer and closer to the first, the gradient of the curve converges to the derivative.



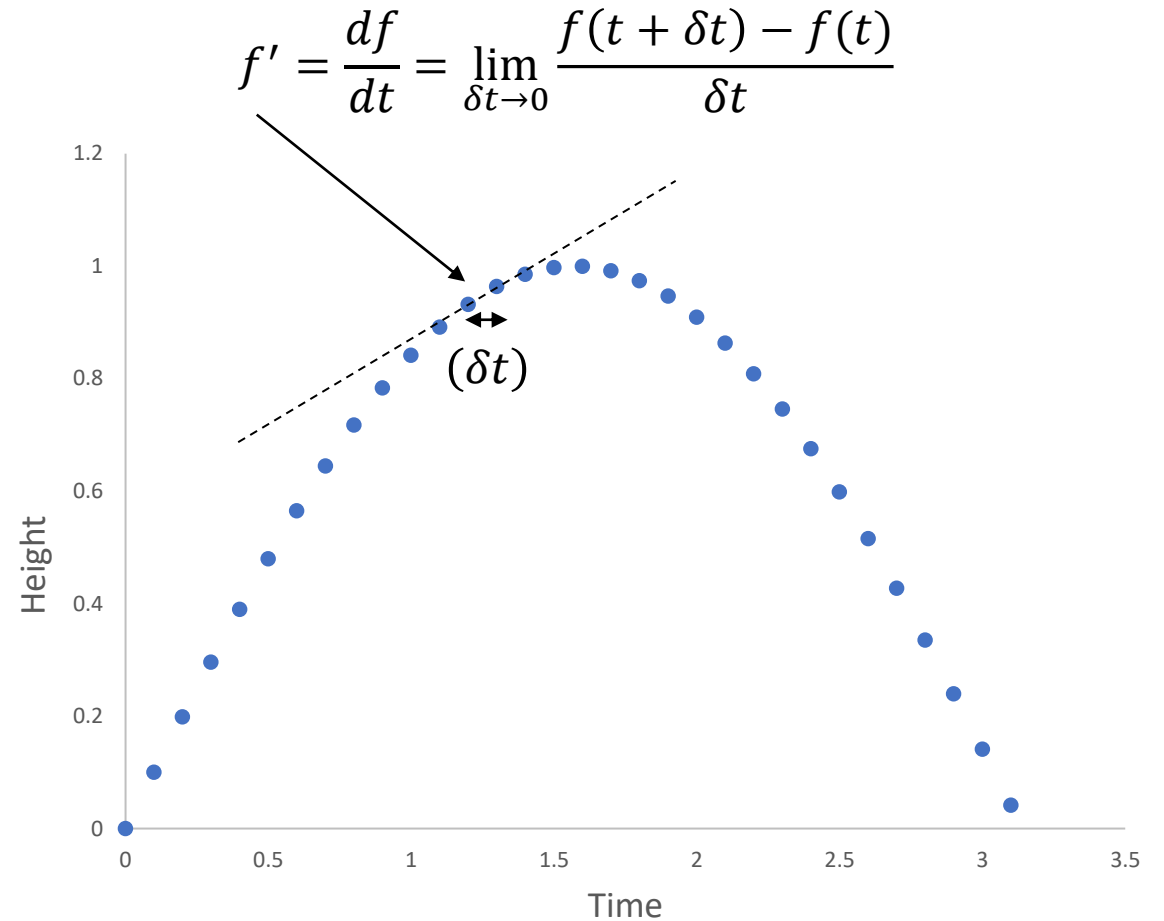
Numerical differentiation

Numerical differentiation rests on the definition of a derivative.

However our data is finite (for example our sensor has a maximum sampling rate) so we can't take the limit of $\delta t \rightarrow 0$

Instead we try to work with very small values of δt

To do this we take our function $f(t + \delta t)$ and we Taylor expand in terms of δt



Taylor series

Lets assume that we know the value of a function $f(t)$ at a given point (call this t) and we also know all of the associated derivatives of the function at that point.

Our task is to approximate the function at near by points.

You have seen this before in P101 with

Prof Booth!

So this is included for revision only...

Taylor expanding $f(t)$ about an arbitrary point t :

$$f(t + \delta t) = f(t) + \delta t \frac{df(t)}{dt} + \frac{\delta t^2}{2!} \frac{d^2 f(t)}{dt^2} + \frac{\delta t^3}{3!} \frac{d^3 f(t)}{dt^3} + \dots,$$
$$= \sum_{n=0}^{\infty} \frac{f^n(t)}{n!} \delta t^n$$

Taylor series revision...

Taylor expand $f(x + a) = \sin(x + a)$ about $a = 0$ (Maclaurin series)

$$f(0) = \sin(0) = 0$$

$$\frac{df(0)}{dx} = \cos(0) = 1$$

$$\frac{d^2f(0)}{dx^2} = -\sin(0) = 0$$

$$\frac{d^3f(x)}{dx^3} = -\cos(0) = -1$$

Taylor expanding $f(x)$ about point $a = 0$:

$$f(x) = f(a) + (x - a) \frac{df(a)}{dx} + \frac{(x-a)^2}{2!} \frac{d^2f(a)}{dx^2} + \frac{(x-a)^3}{3!} \frac{d^3f(a)}{dx^3} + \dots,$$

$$f(x) = 0 + (x - 0) \times 1 + \frac{(x - 0)^2 \times 0}{2!} + \frac{(x - 0)^3 \times (-1)}{3!} + \dots,$$

$$f(x) = x - \frac{x^3}{3!} + \dots,$$

Which is exactly $\sin(x)$!

Forward difference

Recall our definition of the derivative:

$$f' = \frac{df}{dt} = \lim_{\delta t \rightarrow 0} \frac{f(t + \delta t) - f(t)}{\delta t}$$

We can define the **forward difference** approximation as:

$$f'_n \approx \frac{f_{n+1} - f_n}{h} = \frac{f(nh + h) - f(nh)}{h}$$

Where in our case we have a barometer sensor that is sampled at discrete time intervals of length h (5 seconds). Then the value of the function at point n will be given by $f_n = f(nh) = f(5n)$.

Returning to our Taylor series:

$$f(x + \delta x) = f(x) + \delta x \frac{df(x)}{dx} + \frac{\delta x^2}{2!} \frac{d^2 f(x)}{dx^2} + \frac{\delta x^3}{3!} \frac{d^3 f(x)}{dx^3} + \dots,$$

And rearranging we have:

$$\underbrace{\frac{f(x+\delta x) - f(x)}{\delta x}}_{\text{What we want}} = \frac{df(x)}{dx} + \underbrace{\frac{\delta x}{2!} \frac{d^2 f(x)}{dx^2} + \frac{\delta x^2}{3!} \frac{d^3 f(x)}{dx^3} + \dots}_{\text{Error}},$$

Hence the forward difference approximation has an error that will scale with δx – because this is the largest value in the error term!

So the error is “Order” $O(\delta x) = h = \delta t$

“Big O ” notation

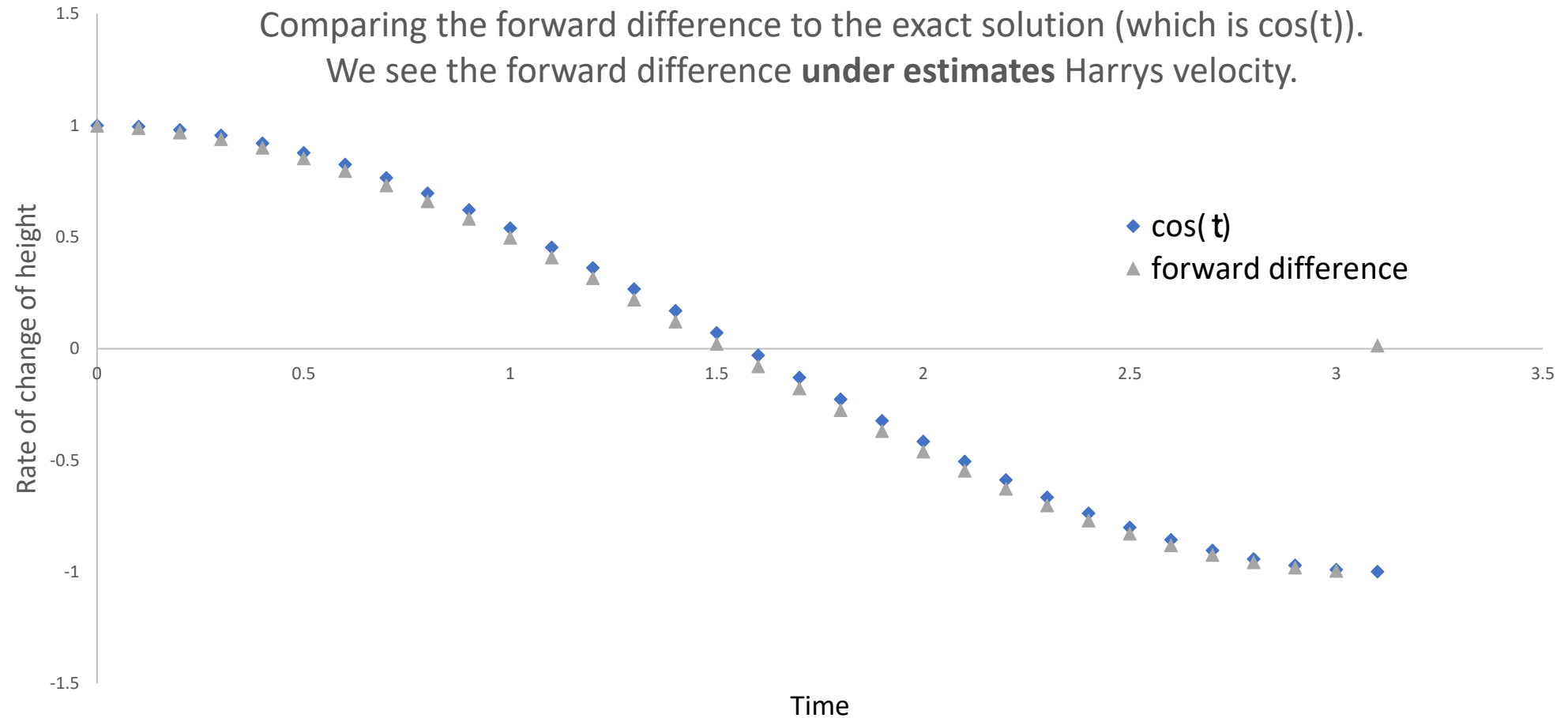
Numerical algorithms are often separated into classes that are defined by “Big O ” notation (sometimes called Landau’s symbol, Bachmann–Landau notation or asymptotic notation), we use this to describe how closely a finite series approximates a given function.

$O(h)$ tells us that our error is proportional to our step size (h).

So if we halve the step size, we would expect the error to halve also.

Notation	Name
$O(1)$	Constant
$O(\log(h))$	Logarithmic
$O((\log(h))^c)$	Polylogarithmic
$O(h)$	Linear
$O(h^2)$	Quadratic
$O(h^c)$	Polynomial
$O(c^h)$	Exponential

Forward difference



Forward difference in more detail

Lets investigate the error a little bit further with some MATLAB...

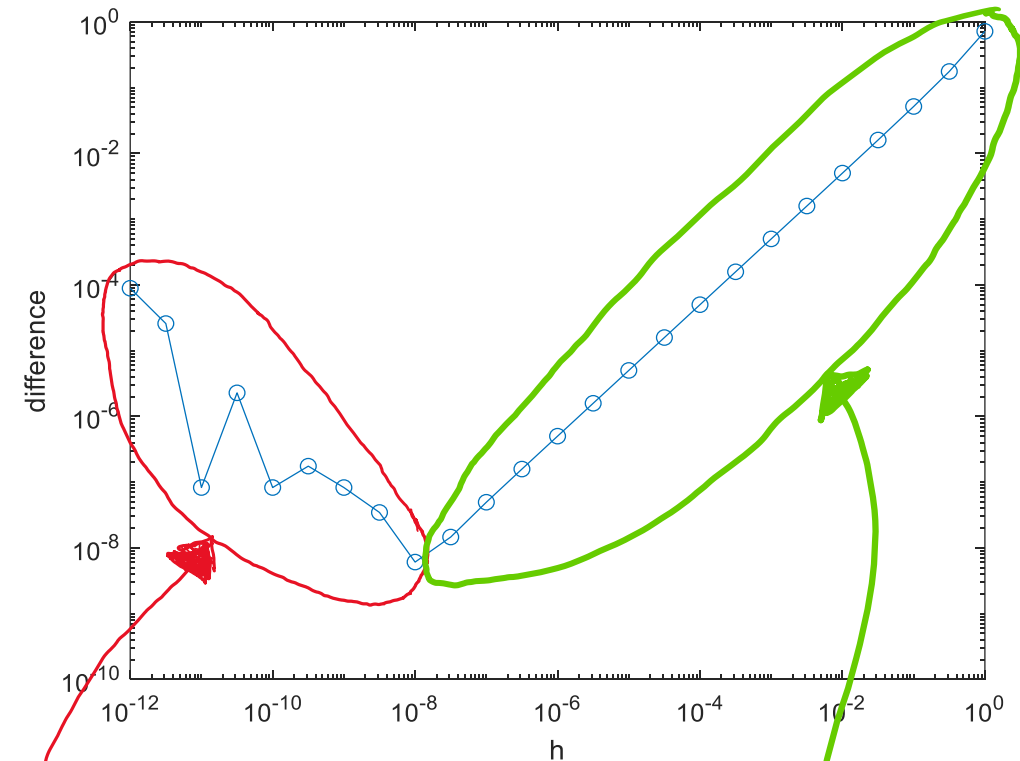
Consider the derivative of $f(x) = e^x$ (it's $f'(x) = e^x$)

At the point $x = 0$ $f(0) = 1$ and so $f'(0) = 1$

Forward difference approximation of the derivative would be:

$$f'_0 \approx \frac{f_{0+h} - f_0}{h} = \frac{f(0 + h) - f(0)}{h} = \frac{e^h - 1}{h}$$

```
h=10.^[0:-0.5:-12];  
exact_derivative=1;  
approx_derivative=(exp(h)-1)./h;  
difference=abs(exact_derivative - approx_derivative);  
figure  
loglog(h,difference,'o-')
```

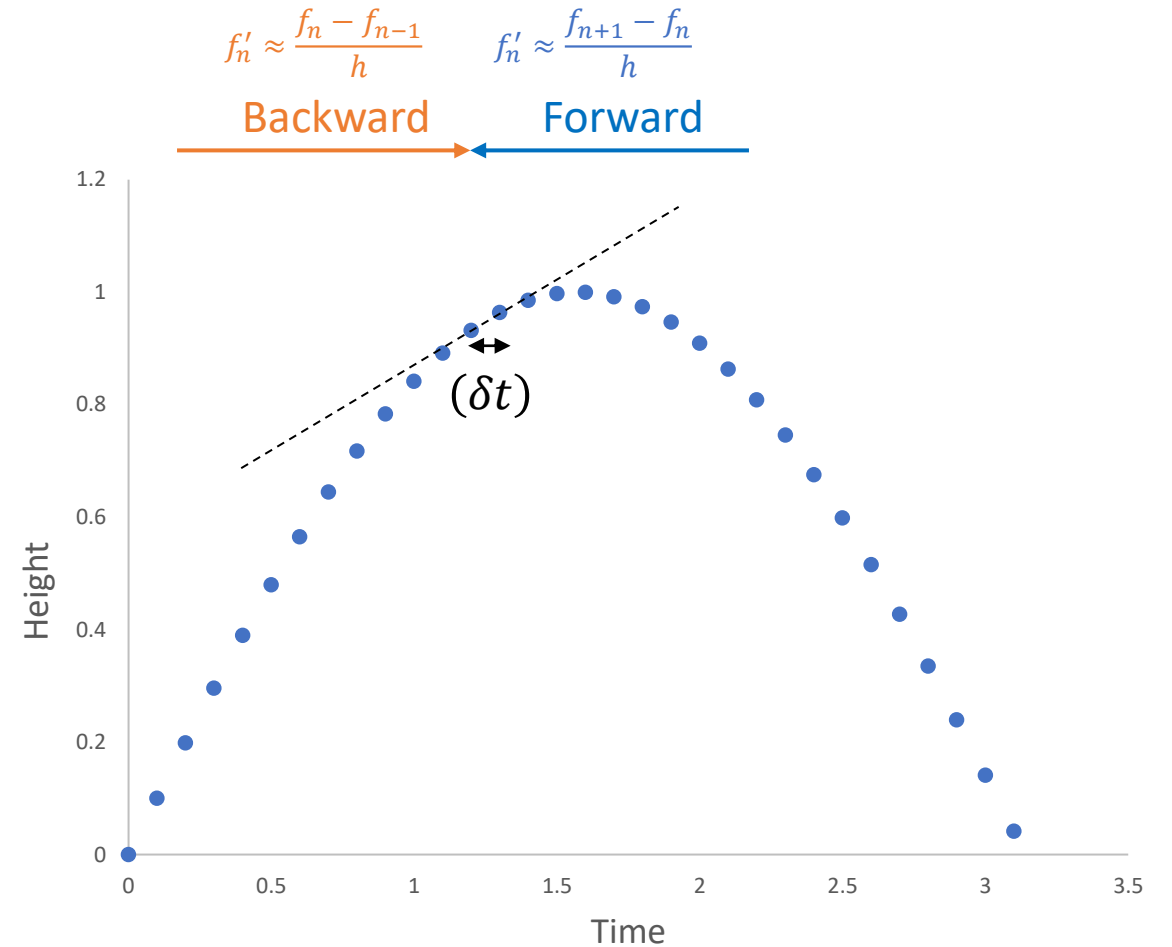


Note that the error behaves as $O(h)$ until **very small h , when rounding errors cause problems.**

Backward difference

The forward difference considers a data point from the right getting closer and closer to the point at which we require our estimate of the derivative.

We could construct a scheme where we have a point from the left getting closer and closer to the point at which we require our estimate of the derivative. This is the backward difference.



Backward difference

The **backward difference** approximation is:

$$f'_n \approx \frac{f_n - f_{n-1}}{h} = \frac{f(nh) - f(nh - h)}{h}$$

Note in the limit $h \rightarrow 0$ we exactly recover our definition of the derivative.

To understand the error of this approximation we need to Taylor expand about $f(x - \delta x)$.

The backward difference is sometimes more useful than the forward difference, even though they have the same error – why?

Returning to our Taylor series:

$$f(x - \delta x) = f(x) - \delta x \frac{df(x)}{dx} + \frac{\delta x^2}{2!} \frac{d^2 f(x)}{dx^2} - \frac{\delta x^3}{3!} \frac{d^3 f(x)}{dx^3} + \dots,$$

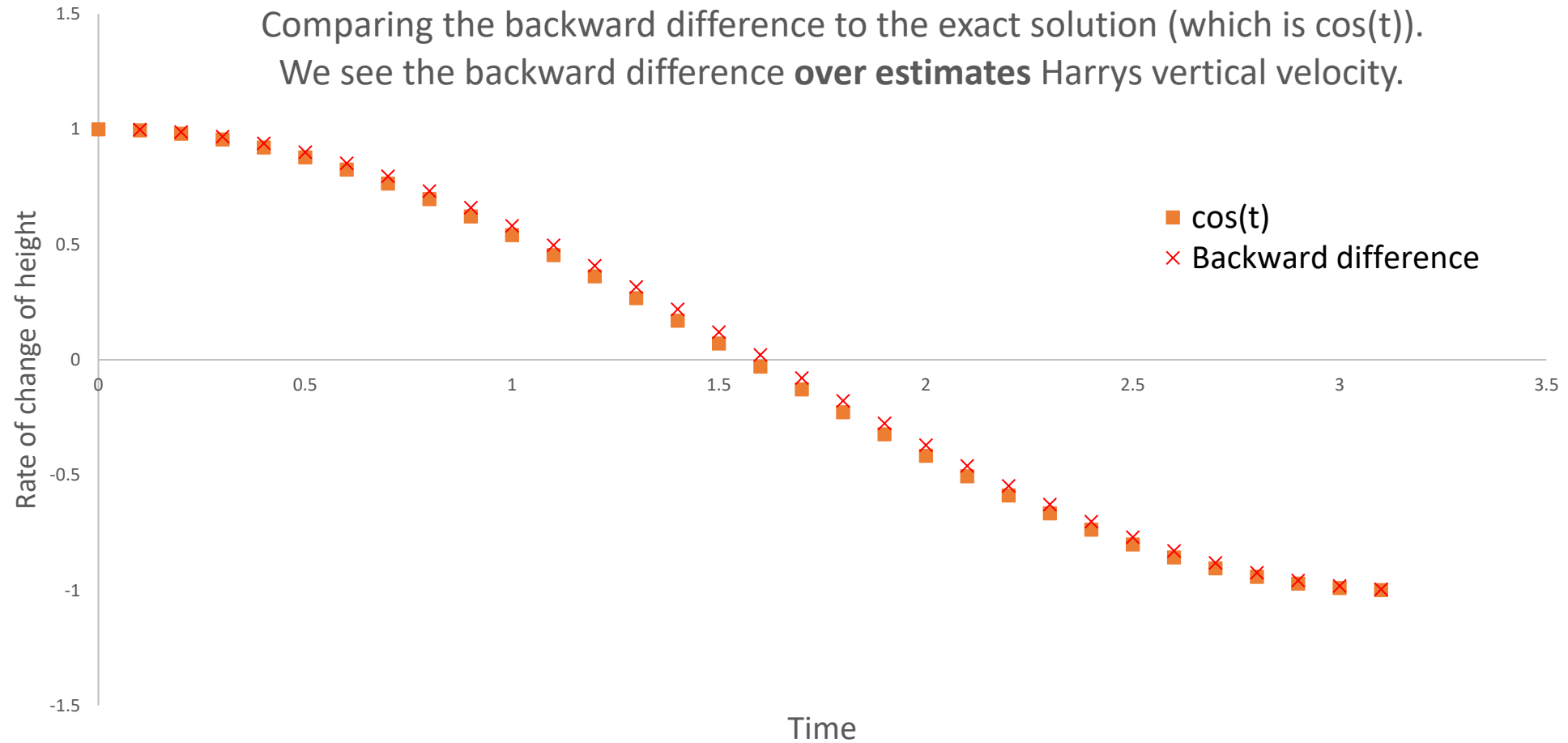
And rearranging we have:

$$\underbrace{\frac{f(x) - f(x - \delta x)}{\delta x}}_{\text{What we want}} = \frac{df(x)}{dx} - \underbrace{\frac{\delta x}{2!} \frac{d^2 f(x)}{dx^2} + \frac{\delta x^2}{3!} \frac{d^3 f(x)}{dx^3} + \dots}_{\text{Error}},$$

Hence the backward difference approximation has an error of order

$$O(\delta x) = h = \delta t$$

Backward difference



Central difference

The **forward difference** approximation is:

$$f'_n \approx \frac{f_{n+1} - f_n}{h} = \frac{f(nh + h) - f(nh)}{h}$$

The **backward difference** approximation is:

$$f'_n \approx \frac{f_n - f_{n-1}}{h} = \frac{f(nh) - f(nh - h)}{h}$$

One overestimates, the other underestimates...

Let's average these together.

$$f'_n \approx \frac{f_{n+1} - f_{n-1}}{2h} = \frac{f(nh + h) - f(nh - h)}{2h}$$

Recall the Taylor series for both f_{n+1} and f_{n-1} :

$$f(x + \delta x) = f(x) + \delta x \frac{df(x)}{dx} + \frac{\delta x^2}{2!} \frac{d^2 f(x)}{dx^2} + \frac{\delta x^3}{3!} \frac{d^3 f(x)}{dx^3} + \dots,$$

$$f(x - \delta x) = f(x) - \delta x \frac{df(x)}{dx} + \frac{\delta x^2}{2!} \frac{d^2 f(x)}{dx^2} - \frac{\delta x^3}{3!} \frac{d^3 f(x)}{dx^3} + \dots,$$

Note: the alternating sign for $f(x - \delta x)$

Central difference.

Then we have:

$$f'_n = \frac{\left(f(x) + \delta x \frac{df(x)}{dx} + \frac{\delta x^2}{2!} \frac{d^2f(x)}{dx^2} + \frac{\delta x^3}{3!} \frac{d^3f(x)}{dx^3} + \dots \right) - \left(f(x) - \delta x \frac{df(x)}{dx} + \frac{\delta x^2}{2!} \frac{d^2f(x)}{dx^2} - \frac{\delta x^3}{3!} \frac{d^3f(x)}{dx^3} + \dots \right)}{2\delta x}$$

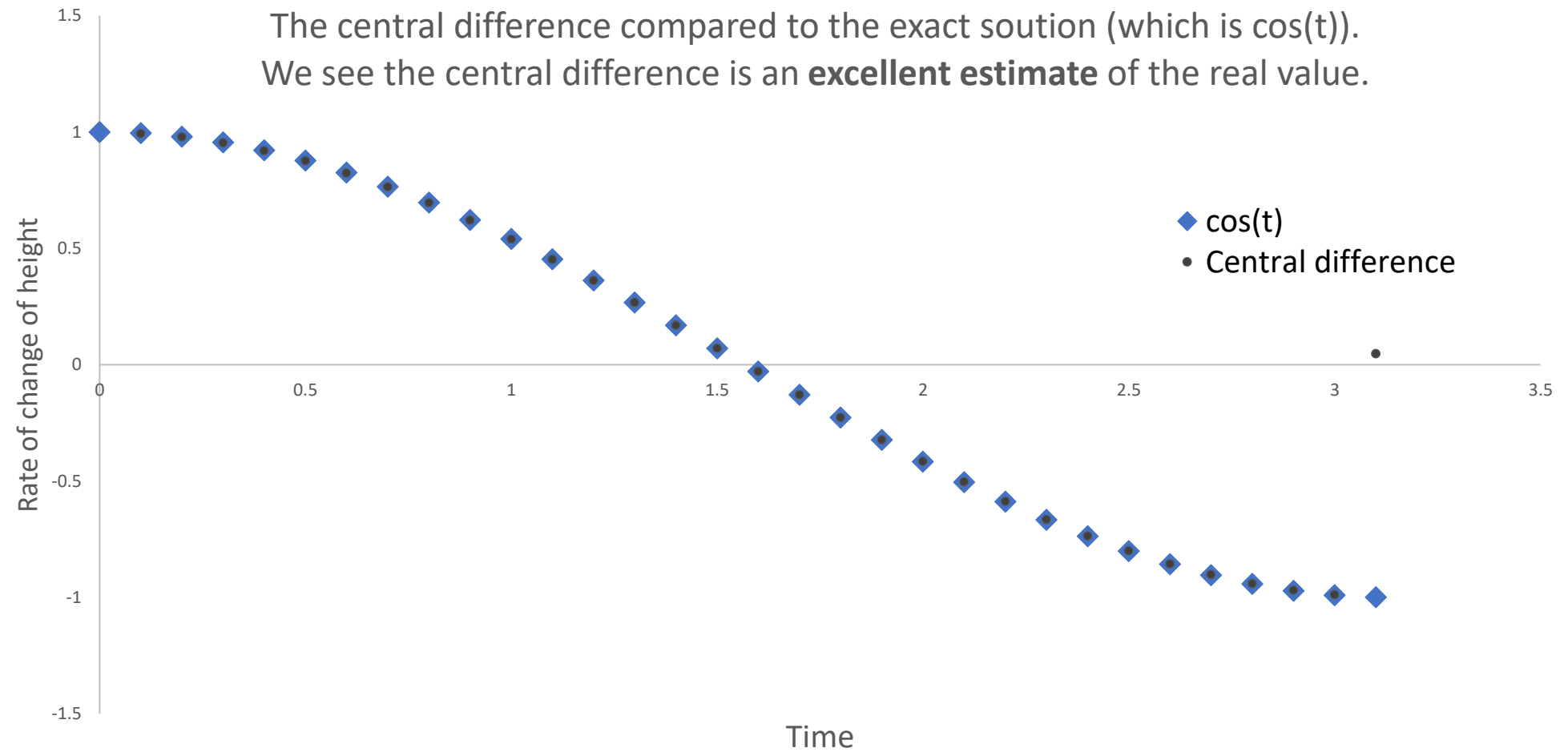
$$f'_n = \frac{2\delta x \frac{df(x)}{dx} + 2 \frac{\delta x^3}{3!} \frac{d^3f(x)}{dx^3} + \dots}{2\delta x}$$

Rearranging:

$$f'_n = \underbrace{\frac{df(x)}{dx}}_{\text{What we want}} + \underbrace{\frac{\delta x^2}{3!} \frac{d^3f(x)}{dx^3} + \dots}_{\text{Error}} .$$

Hence by taking a symmetric approximation we have improved the error from $O(\delta x)$ to $O(\delta x^2)$

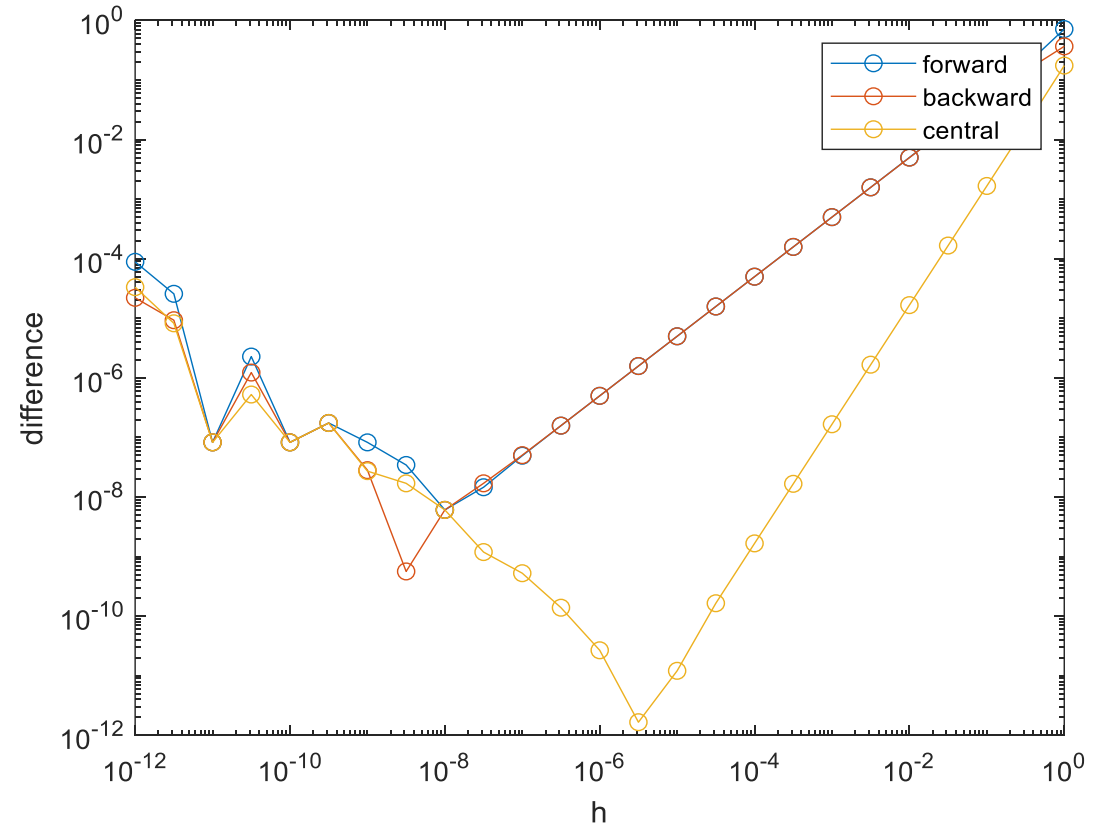
Central difference.



Central difference.

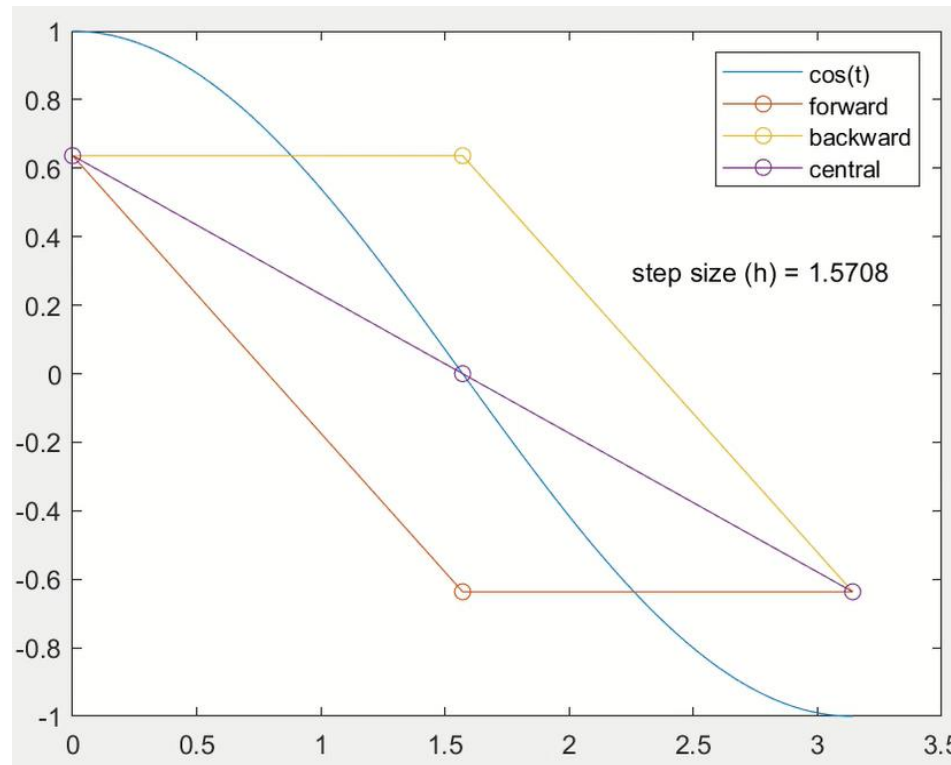
Consider our previous MATLAB example

```
h=10.^[0:-0.5:-12];
exact_derivative=1;
forward_derivative=(exp(h)-1)./h;
backward_derivative=(1-exp(-h))./h;
central_derivative=(exp(h)-exp(-h))./(2*h);
difference_forward=abs(exact_derivative - forward_derivative);
difference_backward=abs(exact_derivative - backward_derivative);
difference_central=abs(exact_derivative - central_derivative);
figure
loglog(h,difference_forward,'-o',h,difference_backward,'-o',h,difference_central,'-o')
legend('forward','backward','central')
xlabel('h')
ylabel('difference')
```



Here we see the error behaving as $O(h^2)$
although again when we have **very small h , rounding errors cause problems.**

Changing the step size (h)



N-point formulae.

The central difference equation is an example of a three-point formula – it gets its name from the fact that it uses a 3x1 neighbourhood about a point.

$$f'_n \approx \frac{f_{n+1} - f_{n-1}}{2h} = \frac{f(nh + h) - f(nh - h)}{2h}$$

You can show that the extended five-point formula

$$f'_n \approx \frac{f_{n-2} - 8f_{n-1} + 8f_{n+1} - f_{n+2}}{12h}$$

Accurate to $O(h^4)$.

Formulae for more than one variable.

It's possible to estimate partial derivatives for more than one variable. Here are some useful formulae:

$$f_x(x, y) = \frac{f(x + h, y) - f(x - h, y)}{2h}$$

$$f_y(x, y) = \frac{f(x, y + k) - f(x, y - k)}{2k}$$

$$f_{xy}(x, y) = \frac{f(x + h, y + k) - f(x + h, y - k) - f(x - h, y + k) + f(x - h, y - k)}{4hk}$$

Formulae for more than one variable.

```
close all; clc

lims=3.5;
figure;

for step=1:-0.05:0.05

    x=-lims:step:lims;
    y=-lims:step:lims;

    [X,Y]=meshgrid(x,y);

    gaussian=exp(-(X.*X+Y.*Y));

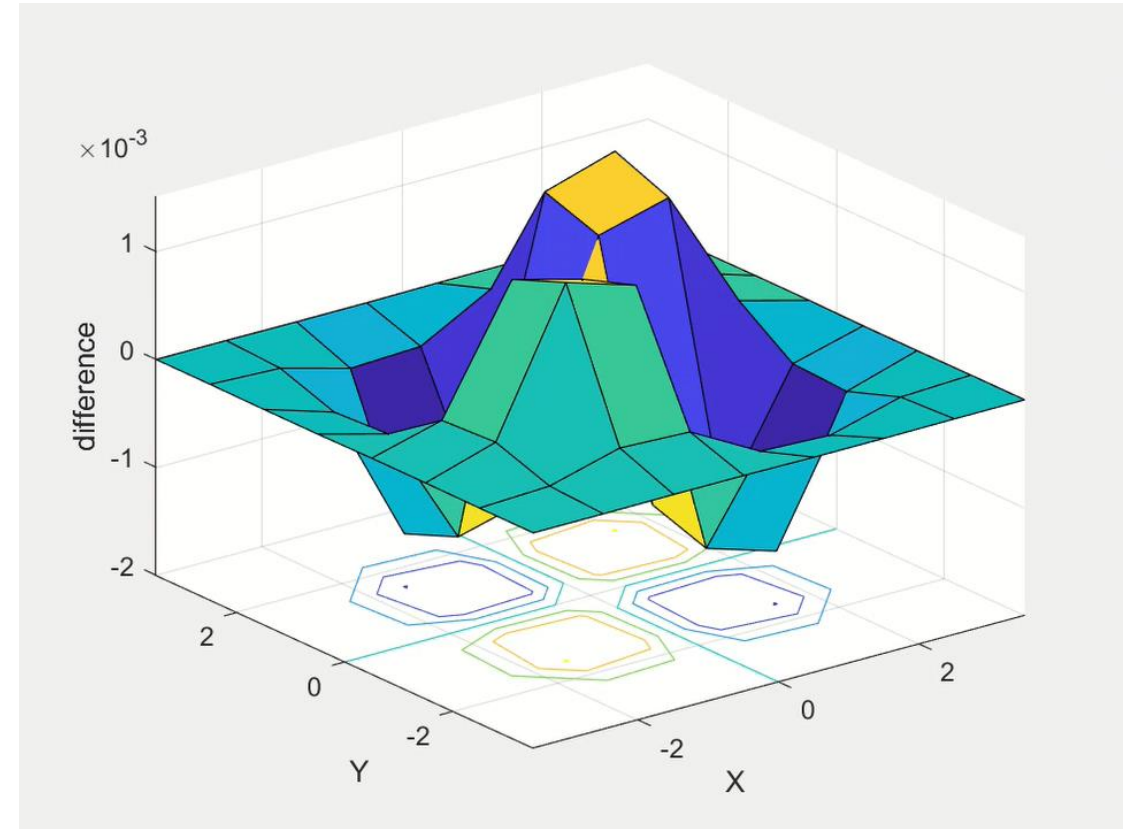
    XPlus=X+step;
    YPlus=Y+step;
    XMinus=X-step;
    YMinus=Y-step;

    difference=(exp(-(XPlus.*XPlus+YPlus.*YPlus)) ...
               -exp(-(XPlus.*XPlus+YMinus.*YMinus)) ...
               -exp(-(XMinus.*XMinus+YPlus.*YPlus)) ...
               +exp(-(XMinus.*XMinus+YMinus.*YMinus))) ...
               /4*step*step;

    surf(X,Y,difference)

    xlabel('X');
    ylabel('Y');
    zlabel('difference');

    drawnow
    pause(step);
end
```



Formulae for higher order derivatives.

It is also possible to construct difference equations that approximate higher order derivatives.

Consider the second order central difference:

$$f''(x) = \frac{\frac{f(x+h) - f(x)}{h} - \frac{f(x) - f(x-h)}{h}}{h} = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2)$$

Found quite simply by applying the central difference scheme to the first derivative...

Using Taylor series expansion (as before) show that the error is $O(h^2)$

Part C – Numerical quadrature

Estimating Integrals

Engineers often need to calculate Integrals approximately, either from data or from functions for which simple analytic forms of the integrals don't exist (*You have seen this in Prelims*).

Some examples of this:

- Calculation of a weight or volume of a body.
- Fuel used from flow rate - time measurements.
- Control systems, e.g. PID (proportional–integral–derivative) controller.
- Integration of a signal over time.

Most methods derive from the basic derivation of (Reimann) integration of a function $f(t)$:

$$y(x) = \int_a^b f(x)dx \rightarrow y(x) = \sum_{m=0}^n a_m f(x_m)$$

With $x_m = mh$ ($m = 0, 1, 2, 3 \dots n$) and a_m being weights.

Rectangular Dissection

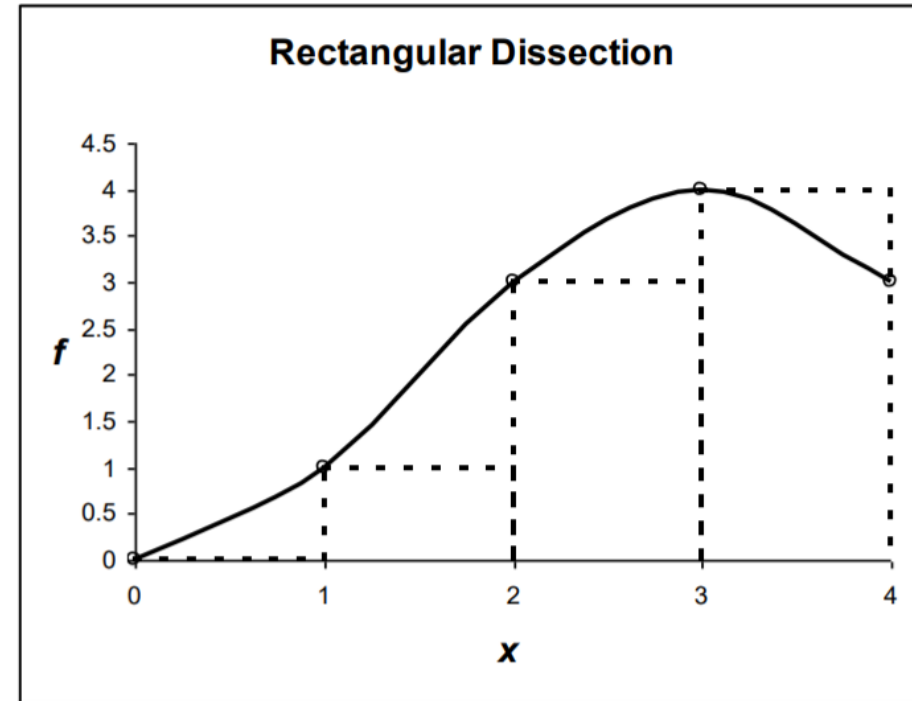
This is the simplest and most “Reimann like” form of numerical integration.

It’s very much like our forward or backward difference schemes.

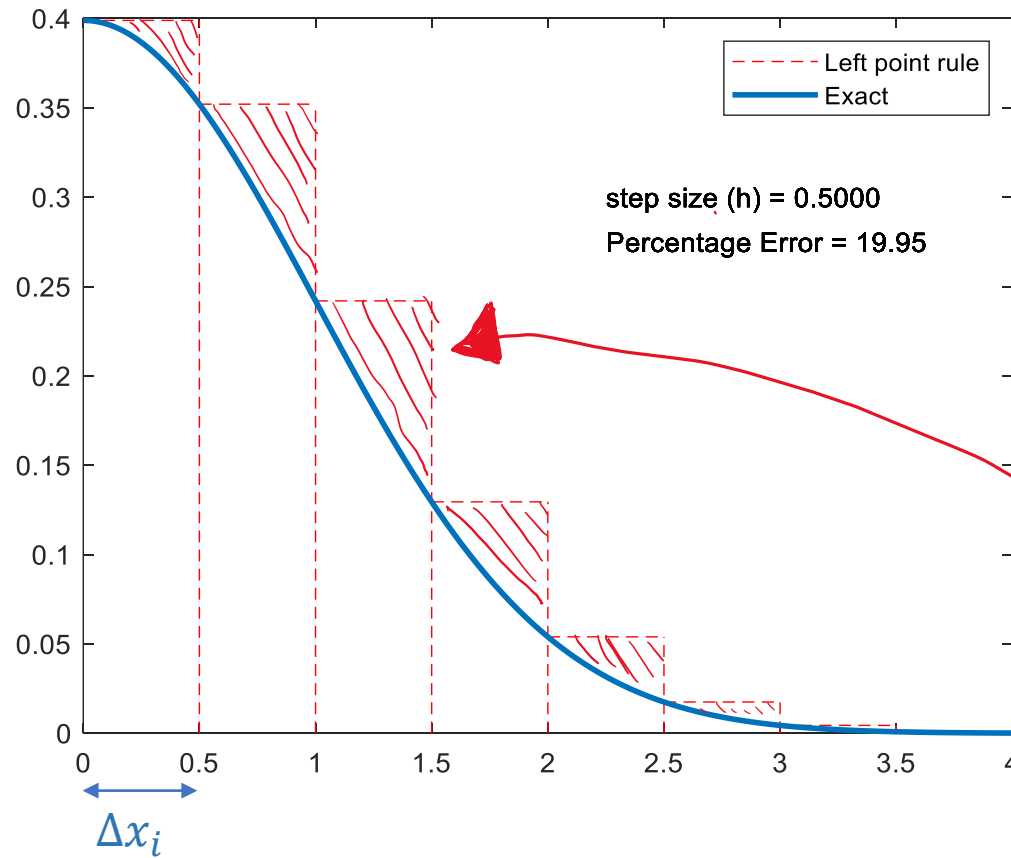
There are three types of Rectangular dissection:

- Left point rule,
- Right point rule,
- Midpoint rule.

Let’s take a look...



The left point rule



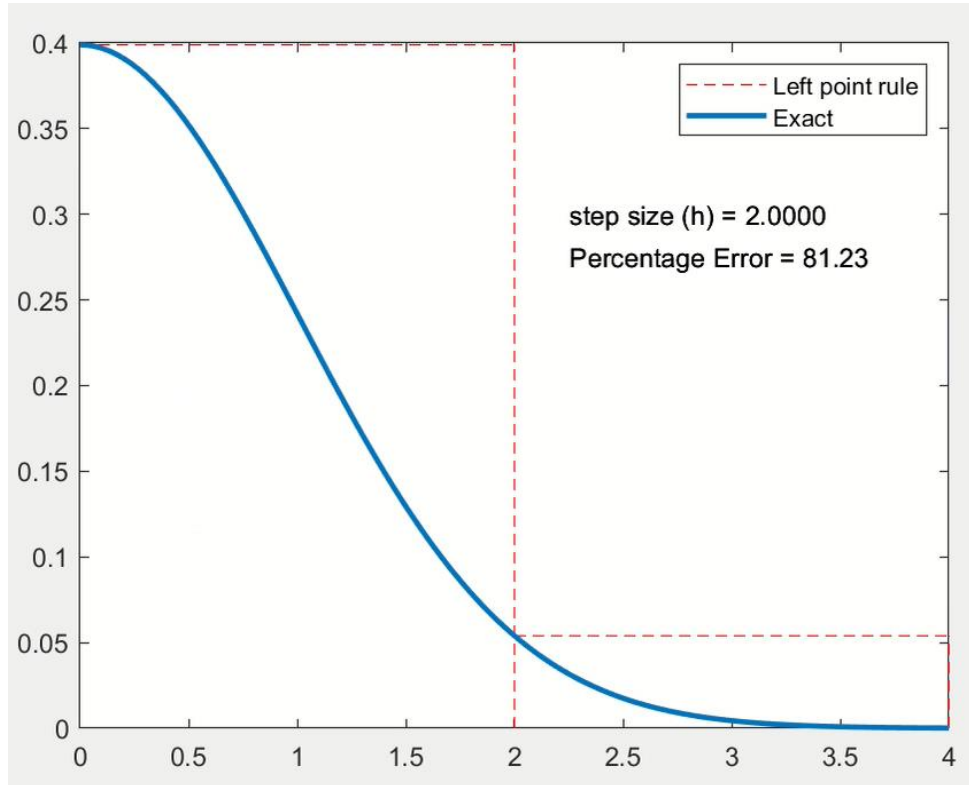
The left point rule is given by the following formula:

$$L_n = \sum_{i=1}^N f(x_{i-1})\Delta x_i$$

Where $\Delta x_i = h$.

Looking at the example on the left we see that the left point rule **over/under estimates** the true value of the integral if **$f(x)$ is (monotonically) decreasing/increasing**.

The left point rule



The left point rule is given by the following formula:

$$L_n = \sum_{i=1}^N f(x_{i-1})\Delta x_i$$

Where $\Delta x_i = h$.

Looking at the example on the left we see that the left point rule **over/under estimates** the true value of the integral if **$f(x)$ is (monotonically) decreasing /increasing**.

Obviously making h smaller decreases the error.

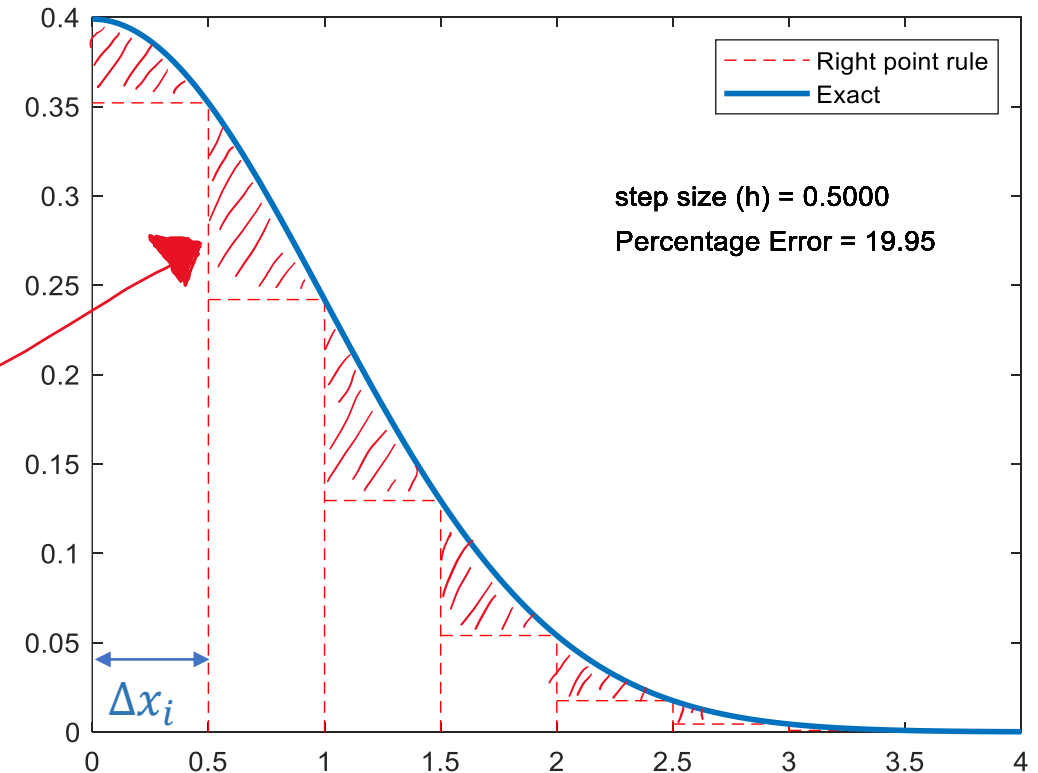
The right point rule

The right point rule is given by the following formula:

$$R_n = \sum_{i=1}^N f(x_i) \Delta x_i$$

Where $\Delta x_i = h$.

Looking at the example on the right we see that the right point rule **under/over estimates** the true value of the integral if **$f(x)$ is (monotonically) decreasing/increasing**.



The right point rule

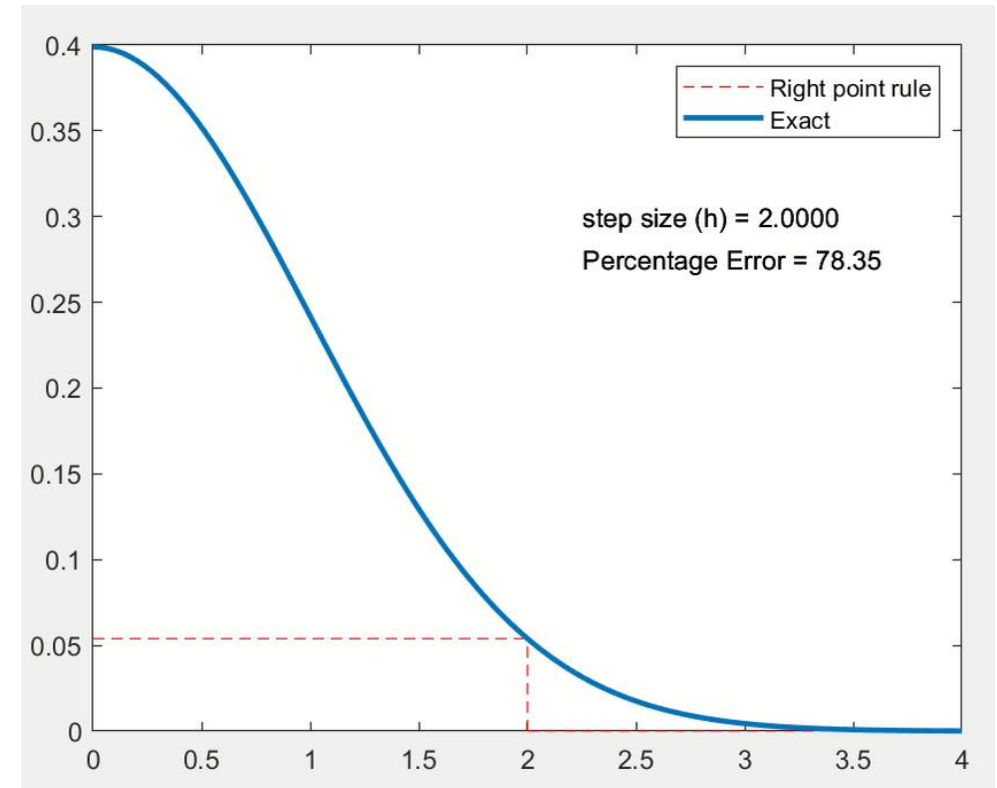
The right point rule is given by the following formula:

$$R_n = \sum_{i=1}^N f(x_i) \Delta x_i$$

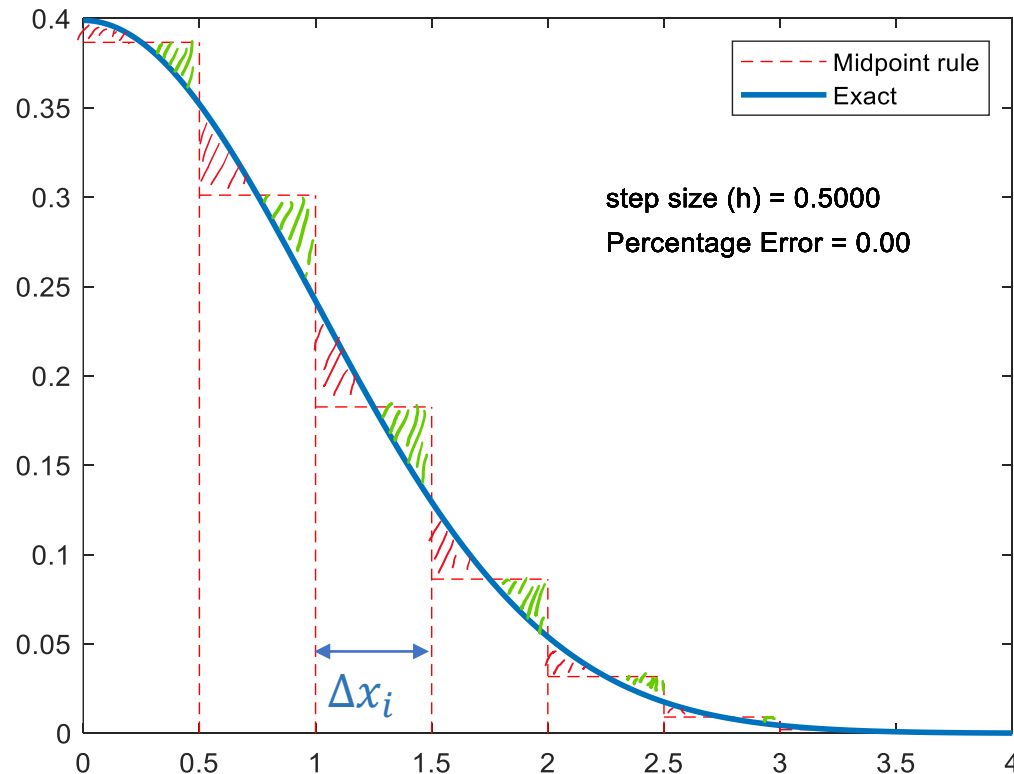
Where $\Delta x_i = h$.

Looking at the example on the right we see that the right point rule **under/over estimates** the true value of the integral if **$f(x)$ is (monotonically) decreasing/increasing**.

Obviously making h smaller decreases the error.



The midpoint rule



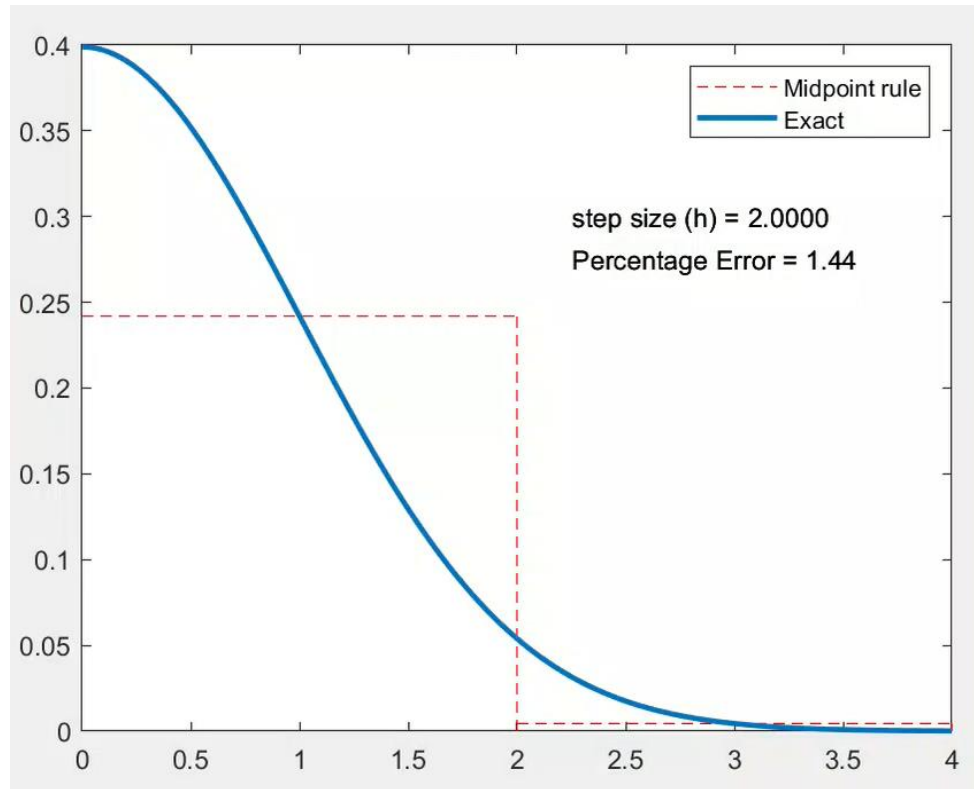
The midpoint rule is given by the following formula:

$$M_n = \sum_{i=1}^N f(x_{i-0.5})\Delta x_i$$

Where $\Delta x_i = h$.

This can help with reducing some error because you typically **over/under** estimate in each rectangular area strip.

The midpoint rule



The midpoint rule is given by the following formula:

$$M_n = \sum_{i=1}^N f(x_{i-0.5})\Delta x_i$$

Where $\Delta x_i = h$.

Not that for this well behaved function (gaussian), the error is near zero once you have just a handful of rectangular areas.

Trapezium rule

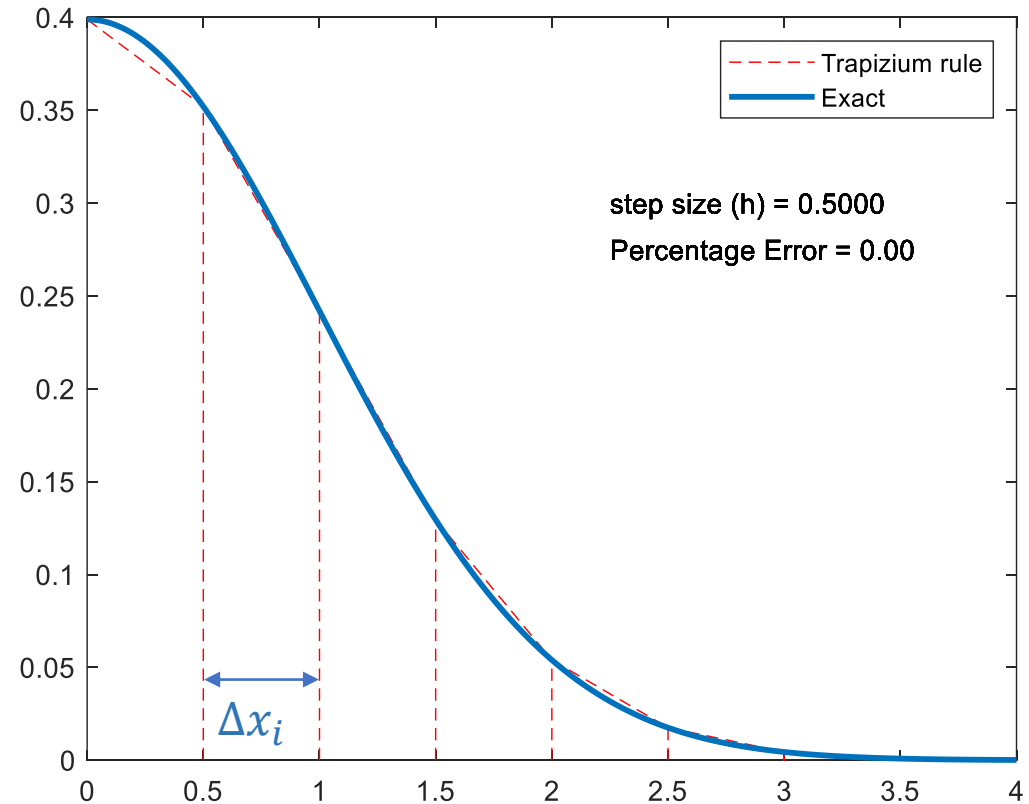
The trapezium rule is better than midpoint. Trapezoids are used in place of rectangular areas, significantly reducing the error for a given integration width (h).

The formula for a the trapezium rule is:

$$T_n = \sum_{i=1}^N \frac{f(x_{i-1}) + f(x_i)}{2} \Delta x_i$$

Where $\Delta x_i = h$.

The plot on the right shows how the error is substantially reduced.

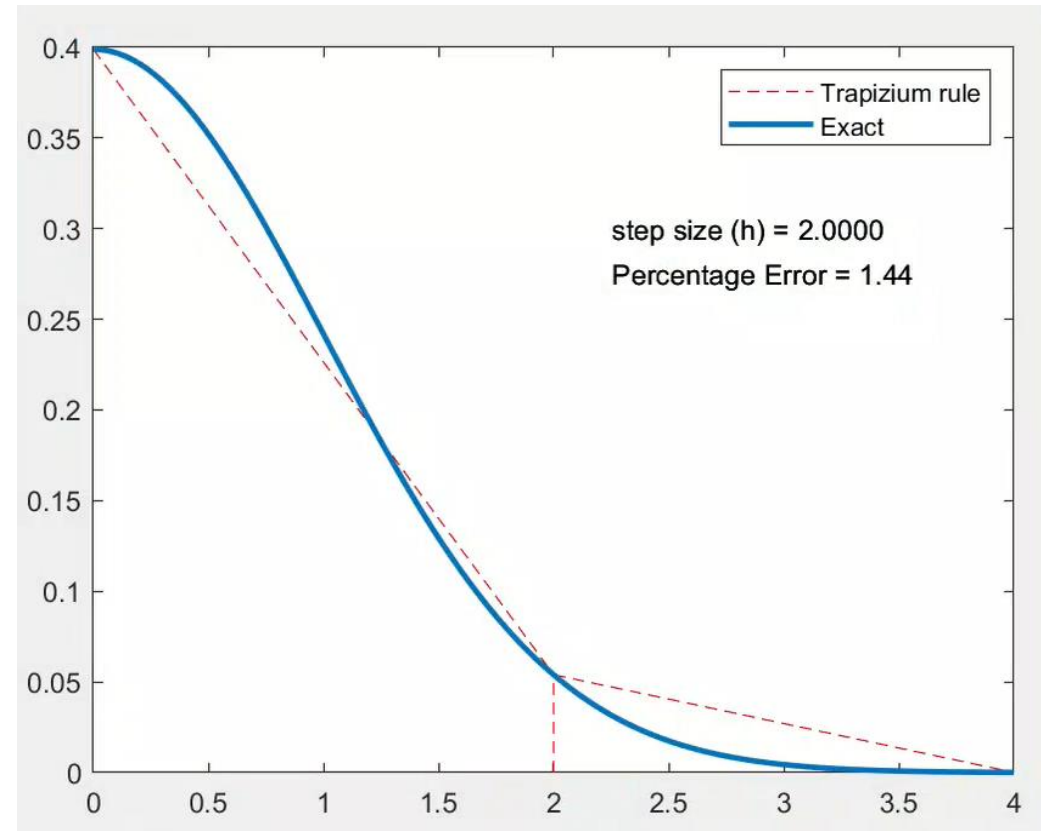


Trapezium rule

It can be shown that the error associated with the trapezium rule is locally $O(h^3)$, the local error is the error per integration strip.

For the full area of integration (so the sum of all of the strips) the algorithm has a global error of $O(h^2)$.

Again let's study the way the error behaves as a function of step size.



Simpson's rule

Fit quadratics (i.e. parabolas) to the middle and end points of an interval $nh \leq x \leq (n+2)h$.

The formula is derived in Kreyszig (p. 960, 7th Ed.) :

$$y(2nh) - y(0) = \frac{h}{3} \{f(0) + 4f(h) + 2f(2h) + 4f(3h) + 2f(4h) + \dots + f(2nh)\}$$

Note the alternating $2f$ and $4f$ terms.

Simpson's rule is **globally accurate to $O(h^4)$** , and is so good that it is not usually necessary to go to more accurate methods.

Kreyszig list an *Algorithm* for integrating by Simpson's rule which could be adapted to MATLAB.

MATLAB has a trapezoidal rule integrator `trapz` and a Simpson's rule integrator `quad`, (short for quadrature)!

Conclusions – what have we learnt?

- Finite difference approximations (numerical differentiation...):
 - formulae for more than one variable,
 - formulae for higher order derivatives.
- Numerical quadrature (also known as numerical integration...):
 - Rectangular dissection,
 - trapezium rule,
 - Simpson's rule,
- Sources of error and convergence:
 - round-off error,
 - truncation error,
 - order of an algorithm,
 - Taylor series analysis.

