***1 - Using the Taylor Expansion to approximate a function***

Mathematically it can be shown that any (well-behaved) function can be reproduced exactly using an infinite sum of a series of polynomial terms:

Here the parameters k0, k1, k2 … are constants that can be relative to the value of the derivatives of *f(t) at t=0.*

Finding the Taylor expansion is useful because we can use it to approximate any function in a small region of interest using only a few simple terms.

To do this we first we choose a starting point ideally in the middle of the region we want to approximate. (The further we move from this starting point the larger the error in our approximation is likely to become.)

The above Taylor expansion adjusted to be centred on a time *t=a* can be written:

where:

We can calculate the values of *k*i by evaluating the value of *f(t)* and its derivatives at time *t=a*.

*k0 = f(a)*

*k1 = f'(a) / 1!*

*k2 = f''(a) / 2!*

*k3 = f’’'(a) / 3!*

…

*kn =* ( *n-th* derivative of *f* at *t=a* ) */ n!*

where:

*n*! means *n* factorial =

*f ’ (t)*  is the first derivative,

*f ’’(t)*  is the second derivative, etc.

***An example – the Taylor expansion of***

To find the Taylor expansion of the exponential function*,*  we first need to examine its derivatives:

In this case the derivative or is also and:

etc.

We can substitute this into the expression for the Taylor expansion of around a:

with coefficients:

!

!

Or in general:

!

This results in:

This can be simplified and written as:

= )

***Truncating the Taylor series***

We can use the infinite sum of the Taylor series to exactly reproduce the function over any time interval. However if we are only interested in the function behaviour close to the expansion point *a,* i.e. for small values of , then a good approximation can still be found if we only use the leading terms of the series.

This is because if is small such that it follows:

e.g if then , =0.001 etc

Therefore in the Taylor expansion if is small, the higher order terms will get smaller and smaller:

If we are only considering a region close to *a* where is small, then the series can be truncated to a few terms and still result in a good approximation, because the first terms dominate the sum and the higher order terms add smaller corrections.

***The “zero-th” order approximation***

If we only include the “zero-th” term we see we are making the approximation:

Note that this has the form of a horizontal line (y = constant).

Here we approximate the function of by a fixed value saying it is approximately equal to the value atfor small .

***The first order approximation***

If we only include the first order term in the expansion we see we are making the approximation:

This has the form of a linear equation y = a + bx. The first order approximation is equivalent to drawing a tangent to the function curve at t=a, and using this line to approximate the function by a straight line.

***The second order approximation***

If we only include the first order term in the expansion we see we are making the approximation:

This has the form of a quadratic equation y = a + bx + cx2. The second order approximation is equivalent to determining the trend in the gradient of the function at point *t=a*, i.e. finding if the gradient getting steeper or shallower. The approximation gives a quadratic function with the same gradient trend, matching the curvature at *t=a*

***Computer Room Tasks 1***

We will look at the truncated Taylor expansion of around a=0

1. Open Canopy and start a new code file. Write a function

get\_factorial(n)

which returns the factorial for a given integer input.

e.g. Check you code using **4!** = 4\*3\*2\*1 = 24

and note that **0!** is defined to be equal to 1 (!!!)

1. Open the file “taylor\_exp.py” in your Canopy workspace.

Read through the code.

Which function generates the ***n***th term in the Taylor expansion?

Which function sums the terms in the Taylor expansion?

1. In the file function get\_factorial(n) is currently empty, add your code from (i) into it to complete the code file.
2. The main function in the file is:

plot\_taylor\_approximation\_exp(n, tmax)

It plots the function that results from a truncated Taylor expansion of the exponential function expanded around t=0, and takes arguments:

n the maximum term used in the Taylor series approximation

tmax the maximum time over which to calculate the approximation

At the bottom of the code you can see a line that calls this function:

plot\_taylor\_approximation\_exp(n=3, tmax=1)

Run the code as it is. Three plots are produced – what are they?

1. Experiment with changing the arguments n and tmax.

Verify that as more terms of the expansion are used the approximation stays accurate (to < 1% difference from the true value) for longer intervals.

To do this:

- Note the times *t* at which the error in the approximation crosses the 1% accuracy line for *n*=1, *n*=2, *n*=3, … using the table below.

- Use this data to plot a figure:

*number of terms* ***n*** vs *time series stays accurate* ***t***.

Hint: You can do this fairly easily if you look at the bottom LHS of the figure window. Here the x,y coordinates of the position of your mouse cursor are shown. You may need to change the argument tmax as appropriate.

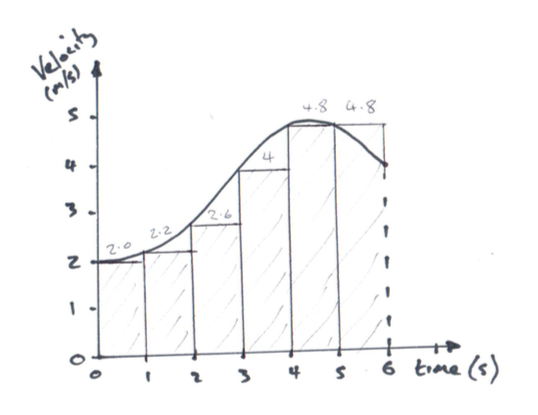
|  |  |
| --- | --- |
| Terms in approximation (*n*) | Accurate to 1% until time *t* (2 d.p.) |
| 0 | 0.01 |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |

***2 - Numerical integration of the area under a curve***

Integration is equivalent to finding the area under the graph of the derivative.

This can be done by numerical methods even if we cannot integrate the expression for the derivative mathematically.

For example the diagram below shows how distance travelled can be estimated from a plot of velocity by breaking the graph into a single of intervals. We find an approximation to the area under the curve by summing the areas of the rectangular strips:



***Computer Room Tasks 2***

i) Open the code in numerical\_integration.py. This integrates the following derivative from t=0 to t=3 by summing the area under the curve as strips.

y'(t) = t3 - 3t2/2 + 4

ii) The correct value for the integral is 18.75. What is minimum number of strips needed to correctly estimate the integral to an accuracy of 1%?

iii) A more accurate method for numerical integration is to use a trapezium
to estimate the area under each strip. This is coded but commented out.
Uncomment the lines to enable this approximation.

iv) What is minimum number of trapezoid strips needed to correctly estimate the integral to an accuracy of 1%

***3 - Developing a function to solve ODEs***

There are many different methods or ‘algorithms’ for numerically solving an ODE of the general form:

In this task we will develop our own code to solve an ODE

***The Forward Euler Method***

The Forward Euler method uses the value of dy/dt to determine how the value of y will change over an interval. It does this by breaking the interval into a series of small timesteps and applying the following procedure.

Here we have used the following:

* **y’ = f(y,t).** This is the ODE that we are trying to solve, describing the rate of change dy/dt for any given values of *y* and *t*.
* **y0.**This is the initial condition which is the value of *y* at a given time *t0*.
* . This is the size of the time step used by the algorithm.
* This denotes the calculated value of *y* at time

***How it works:***

We start at time *t0* and use the initial condition that gives us *y0* i.e. the value of *y* at *t0*. We then use the following calculation to move forward in time one step at a time.

Here the time points used are written as :

i.e.

e.g. after 1 step we are at time

The values we calculated for *y* at each timepoint are written as yn.

e.g.

is the value we calculated *y* at time

We start by calculating the value *y* after 1 timestep from :

We then use the value to find the value after a second timestep:

We can continue this process until we have covered the time interval we are interested in.

***An Example: Population Dynamics***

Let’s consider a population of size P in which birth and death rates per individual stay constant as time passes (even if population levels are changing).

In this case the total number of births in any given time will be proportional to the size of the population:

***total number of births per unit time P***

Similarly:

***total number of deaths per unit time P***

And so:

***net change (births minus deaths) in P per unit time P***

Mathematically we can write this as:

Or in the form of a differential equation:

Here we have used *y* to represent the population size, and *r* is the constant of proportionality, which is related to the rate of net change for the population per individual.

***Computer Room Tasks 3***

i) First we will carry out the Forward Euler method above manually to numerically integrate the differential equation we developed for population dynamics:

First we will complete the process manually, using:

timestep: = 0.1

*parameters*

net rate of population change: r = -0.5

initial condition: y0 = 10,000 at time t0 = 0

To do this fill out the following table. (Use Python as your calculator!)

The first calculation is below:

where we have used

|  |  |  |
| --- | --- | --- |
| ***n*** | ***tn*** | ***yn*** |
| 0 | 0 | **10000.0** |
| 1 | 0.1 | **9,500** |
| 2 | 0.2 |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |

Hint. Try to use variables to store the parts of the calculations you do, this will mean you can do calculations for subsequent terms more easily.

ii) Write some Python code to perform this algorithm using a loop. Use it to calculate up to time = 2 in time steps of 0.1.

Use lists to store each value of and you calculate.

iii) Plot the graph of the approximation you have made.

iv) Find the exact solution to the problem mathematically. Plot this function on your graph using a dotted line.

How does the numerical approximation compare with to the exact solution?

v) Adjust your code so that you can change the following parameters:

a) the value of *r*.

Verify for positive values of *r* you see exponential growth.

b) the value of

Verify that for smaller step sizes the approximation becomes more accurate.

c) the initial condition y0

Note: depending on how you have written your code you may need to update the formula used to calculate the exact solution to reflect the change in initial condition.

vi) When *r* is negative and the step size is made too wide, the method can become unstable. Try to reproduce this.

(Hint: this effect can be seen most clearly when the step size is close to 2/|*r*|).

vii) Can you verify that for even very large values of *r*, an accurate approximation can be made if the step size is small enough?

What step size is necessary to be able to approximate the system when *r =* -100, and the system starts with initial condition *y0*=1000 at *t0*=0.

viii) Change the expression used for the derivative so that it has the form:

Use initial conditions *y0* = 100 at t=0, with parameters *r* = 0.5 and *K* = 1000 and a small step size. How has the behaviour of *y(t)* changed?

Explore how changing *y0* affects the system. In particular what is the behaviour when:

*a) y* is much smaller than *K*

*b) y* is much larger than *K*

*c) y* is close to *K*

What is the meaning of parameter *K*?

ix) Reorganise your code so that the numerical solution to our ODE can be run as a single function that takes five arguments:

,

Where and are the start and end times over which to integrate*.*

It should return two lists of the calculated *yn* and *tn* values.

Hint. The following example can be used to return multiple items from a function:

def test\_function():

mylist1=[0,1,2,3,4,5]

mylist2=[2,3,9,1,8]

return [ mylist1, mylist2 ]

results=test\_function()

outlist1=results[0]

outlist2=results[0]

print outlist1

print outlist2

x) ***Optional*** In future sessions we will use a more sophisticated ODE solver from a Python module. This takes a form like

**odeint ( ydot , y0, [t\_start, t\_end], args=[r] ,stepsize)**

where:

**ydot** is a user defined function (see below)

**y0** is the initial value of y at time t\_start

**t\_start** is the starting time to be used

**t\_end** is the time at which to stop the calculation

**args** is a list of any parameters which are used to

calculate the derivative

**stepsize** is the size of the time steps to be used

Before using the odeint function the user must define a function ydot.

This is a function that returns the derivative in our ODE, e.g. If then the function would look like this:

def ydot(y, t, args):

r=args[0]

dydt = r\*y

return dydt

Try to reorganize your code so that it has the above form, consisting of:

a) a function ydot as given above

b) a function odeint that performs the integration