Sequences and Series

Contents

Mathematics Methods 1

```
# This cell just imports relevant modules

import numpy
from sympy import sin, cos, exp, ln, Function, Symbol, diff, integrate, limit, oo,
series, factorial
from math import pi
import mpmath
import matplotlib.pyplot as plt
```

Sequences

Slide 10

```
# Store elements of the sequence in a list
finite_sequence = [2*k for k in range(1, 5)]

print(finite_sequence)
```

```
[2, 4, 6, 8]
```

Remember: `range(A,B)` generates integers from `A` up to `B-1`, so we need to use `B=5` here.

Convergence of sequences

Slides 11, 13, 14

```
k = Symbol('k')
print("As k->infinity, the sequence {k} tends to:", limit(k, k, oo))
# The 'oo' here is SymPy's notation for infinity

print("As k->infinity, the sequence {1/k} tends to:", limit(1.0/k, k, oo))

print("As k->infinity, the sequence {exp(1/k)} tends to:", limit(exp(1.0/k), k, oo))

print("As k->infinity, the sequence {(k**3 + 2*k - 4)/(k**3 + 1)} tends to:",
      limit((k**3 + 2*k - 4)/(k**3 + 1), k, oo))
```

```
As k->infinity, the sequence {k} tends to: oo
As k->infinity, the sequence {1/k} tends to: 0
As k->infinity, the sequence {exp(1/k)} tends to: 1
As k->infinity, the sequence {(k**3 + 2*k - 4)/(k**3 + 1)} tends to: 1
```

Series

Slide 15

```
# Using list comprehension:
print("The sum of 3*k + 1 (from k=0 to k=4) is:", sum([3*k + 1 for k in range(0,5)]))
# Note: we could also use the nsum function (part of the module mpmath):
# print(mpmath.nsum(lambda k: 3*k + 1, [0, 4]))
```

```
The sum of 3*k + 1 (from k=0 to k=4) is: 35
```

```
x = 1
print(f"The sum of (x**k)/(k!) from k=0 to k=4, with x = {x}, is:",
      sum([x**k/factorial(k) for k in range(1,5)]))
```

```
The sum of (x**k)/(k!) from k=0 to k=4, with x = 1, is: 41/24
```

Arithmetic progression

Slide 18

```
print("The sum of 5 + 4*k up to the 11th term (i.e. up to k=10) is:",
      sum([5 + 4*k for k in range(0,11)]))
```

```
The sum of 5 + 4*k up to the 11th term (i.e. up to k=10) is: 275
```

Geometric progression

Slide 21

```
print("The sum of 3**k up to the 7th term (i.e. up to k=6) is:",
      sum([3**k for k in range(0,7)]))
```

```
The sum of 3**k up to the 7th term (i.e. up to k=6) is: 1093
```

Infinite series

Slides 23, 24, 25

```
print("The sum of the infinite series sum(1/(2**k)) is:",
      mpmath.nsum(lambda k: 1/(2**k), [1, mpmath.inf]))
print("The sum of the infinite alternating series sum(((-1)**(k+1))/k) is:",
      mpmath.nsum(lambda k: ((-1)**(k+1))/k, [1, mpmath.inf]))
```

```
The sum of the infinite series sum(1/(2**k)) is: 1.0
The sum of the infinite alternating series sum(((-1)**(k+1))/k) is: 0.693147180559945
```

Ratio test

Slide 27

A diverging example:

```
k = Symbol('k')
f = (2**k)/(3*k)
f1 = (2**(k+1))/(3*(k+1))
ratio = f1/f

lim = limit(ratio, k, oo)
print("As k -> infinity, the ratio tends to:", lim)
if(lim < 1.0):
    print("The series converges")
elif(lim > 1.0):
    print("The series diverges")
else:
    print("The series either converges or diverges")
```

```
As k -> infinity, the ratio tends to: 2
The series diverges
```

A converging example:

```
f = (2**k)/(5**k)
f1 = (2**(k+1))/(5**(k+1))
ratio = f1/f

lim = limit(ratio, k, oo)
print("As k -> infinity, the ratio tends to:", lim)
if(lim < 1.0):
    print("The series converges")
elif(lim > 1.0):
    print("The series diverges")
else:
    print("The series either converges or diverges")
```

```
As k -> infinity, the ratio tends to: 2/5
The series converges
```

Power series

Slide 30

```
k = Symbol('k')
x = Symbol('x')

a = 1/k
f = a*(x**k)

a1 = 1/(k+1)
f1 = a1*(x**(k+1))

ratio = abs(a/a1)
R = limit(ratio, k, oo)
print("The radius of convergence (denoted R) is:", R)

x = 0.5
if(abs(x) < 1):
    print(f"The series converges for |x| = {abs(x)} (< R)")
elif(abs(x) > 1):
    print(f"The series diverges for |x| = {abs(x)} (> R)")
else:
    print(f"The series either converges or diverges for |x| = {abs(x)} (== R)\n")
```

```
The radius of convergence (denoted R) is: 1
The series converges for |x| = 0.5 (< R)
```

Useful series

Slide 34

```
x = Symbol('x')
r = Symbol('r')

# Note: the optional argument 'n' allows us to truncate the series
# after a certain order of x has been reached.
print("1/(1+x) = ", series(1.0/(1.0+x), x, n=4))
print("1/(1-x) = ", series(1.0/(1.0-x), x, n=4))
print("ln(1+x) = ", series(ln(1.0+x), x, n=4))
print("exp(x) = ", series(exp(x), x, n=4))
print("cos(x) = ", series(cos(x), x, n=7))
print("sin(x) = ", series(sin(x), x, n=8))
```

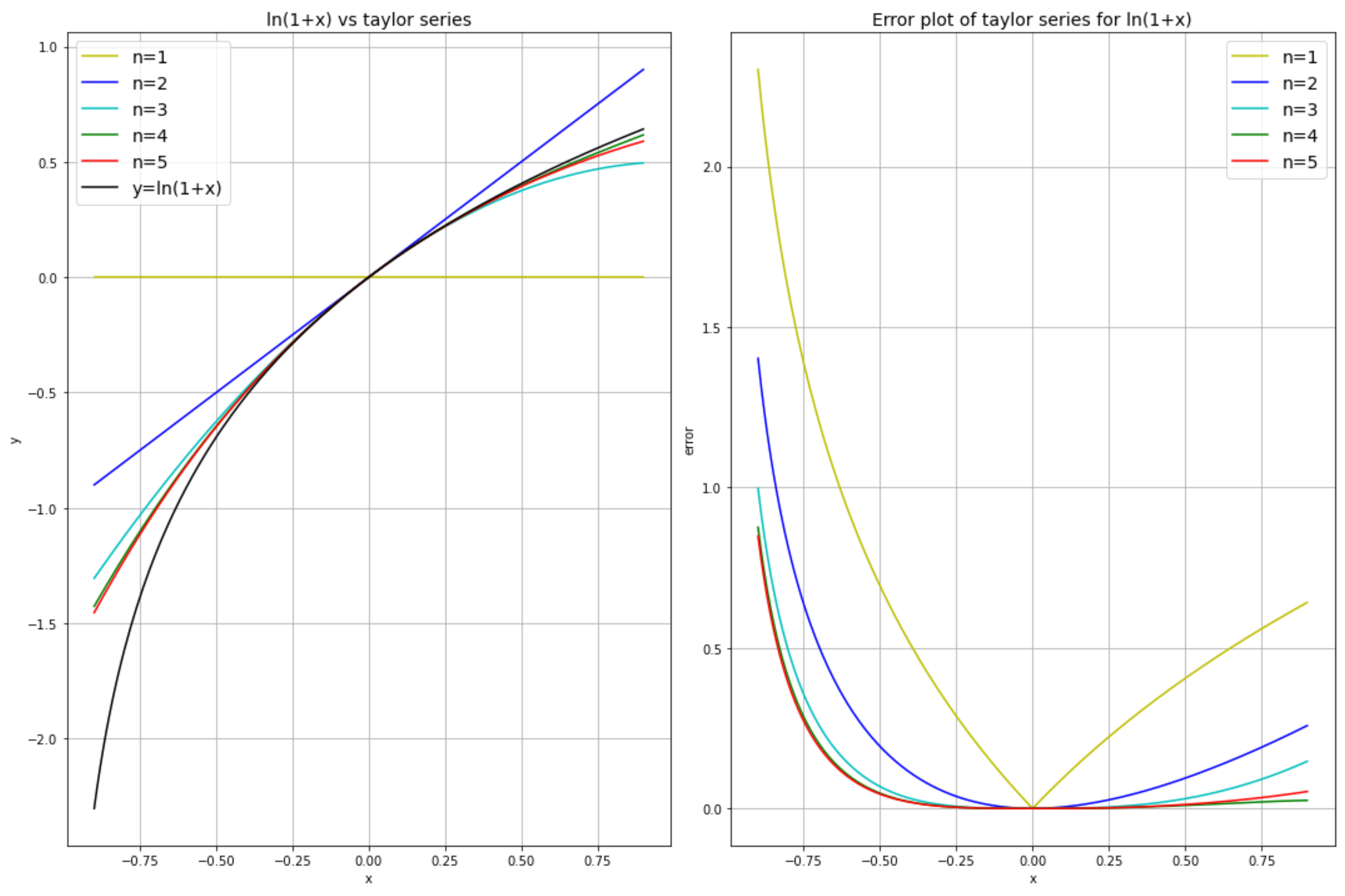
```
1/(1+x) =  1.0 - 1.0*x + 1.0*x**2 - 1.0*x**3 + O(x**4)
1/(1-x) =  1.0 + 1.0*x + 1.0*x**2 + 1.0*x**3 + O(x**4)
ln(1+x) =  1.0*x - 0.5*x**2 + 0.333333333333333*x**3 + O(x**4)
exp(x) =  1 + x + x**2/2 + x**3/6 + O(x**4)
cos(x) =  1 - x**2/2 + x**4/24 - x**6/720 + O(x**7)
sin(x) =  x - x**3/6 + x**5/120 - x**7/5040 + O(x**8)
```

Taylor series

The more terms there are in the series the more accurate it is as it better approximates the function. The error of the Taylor series decreases as x approaches 0.

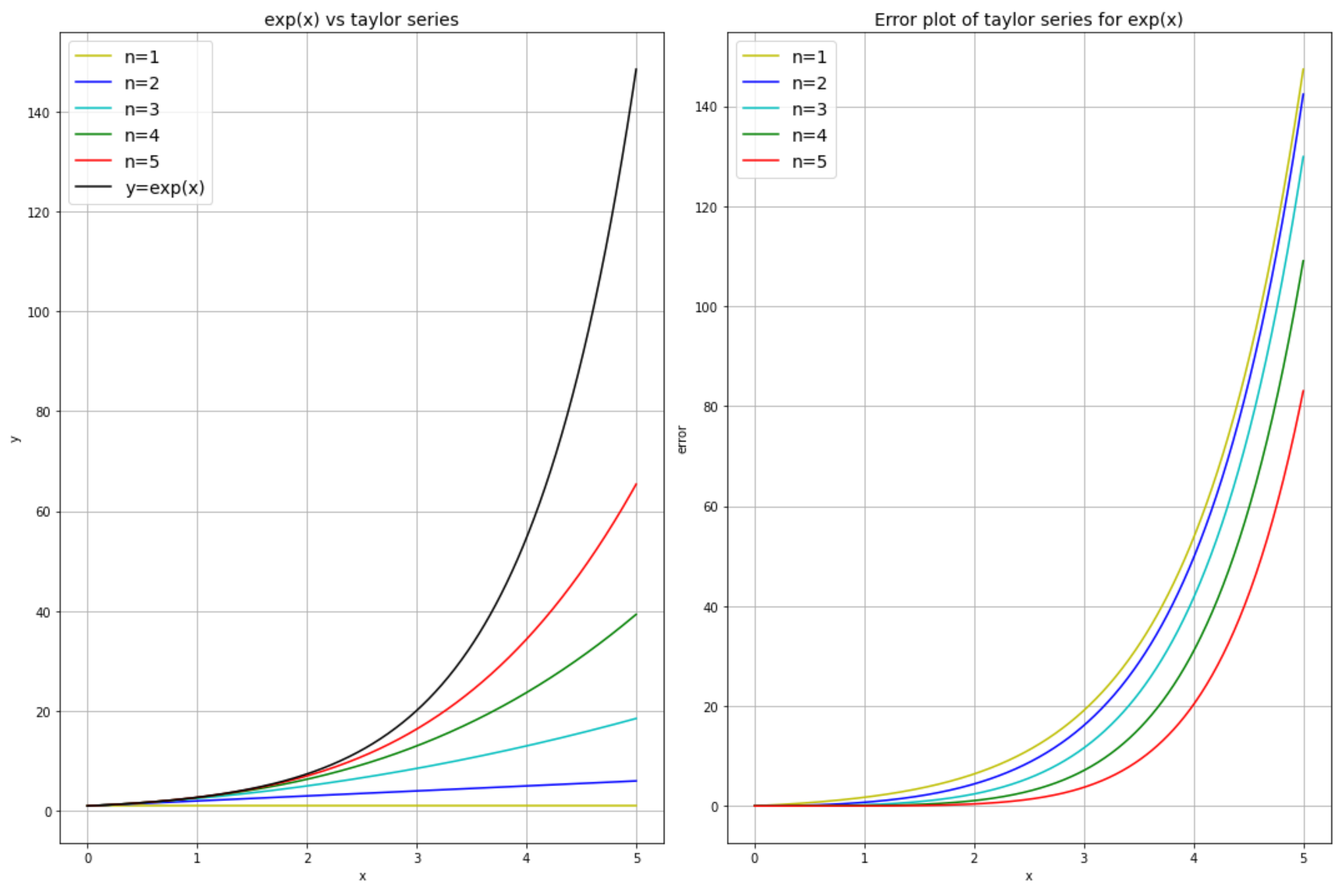
$\ln(1+x)$ with different number of terms n:

```python
def ln_taylor(x, n):
    y = numpy.zeros(x.shape)
    for i in range(1, n):
        y = y + (-1)**(i+1) * (x**i)/(factorial(i))  # taylor series of ln(1+x)
    return y


x = numpy.linspace(-0.9, 0.9, 181)  # [-0.9, -0.89, ..., 0.88, 0.89, 0.9]

ln = numpy.log(1+x)  # actual function

n = [i for i in range(1, 6)]  # list [1, 2, 3, 4, 5]
colour = ['y', 'b', 'c', 'g', 'r']

fig, ax = plt.subplots(1, 2, figsize=(15, 10))

for i in range(len(n)):
    y_ln = ln_taylor(x, n[i])  # values from taylor series
    ln_error = abs(y_ln - ln)  # difference between taylor series and function

    ax[0].plot(x, y_ln, colour[i], label=f'n={n[i]}')
    ax[1].plot(x, ln_error, colour[i], label=f'n={n[i]}')

ax[0].plot(x, ln, 'k', label='y=ln(1+x)')

ax[0].set_ylabel('y')
ax[0].set_title('ln(1+x) vs taylor series', fontsize=14)
ax[1].set_ylabel('error')
ax[1].set_title('Error plot of taylor series for ln(1+x)', fontsize=14)
for i in range(len(ax)):
    ax[i].set_xlabel('x')
    ax[i].legend(loc='best', fontsize=14)
    ax[i].grid(True)

fig.tight_layout()
plt.show()
```



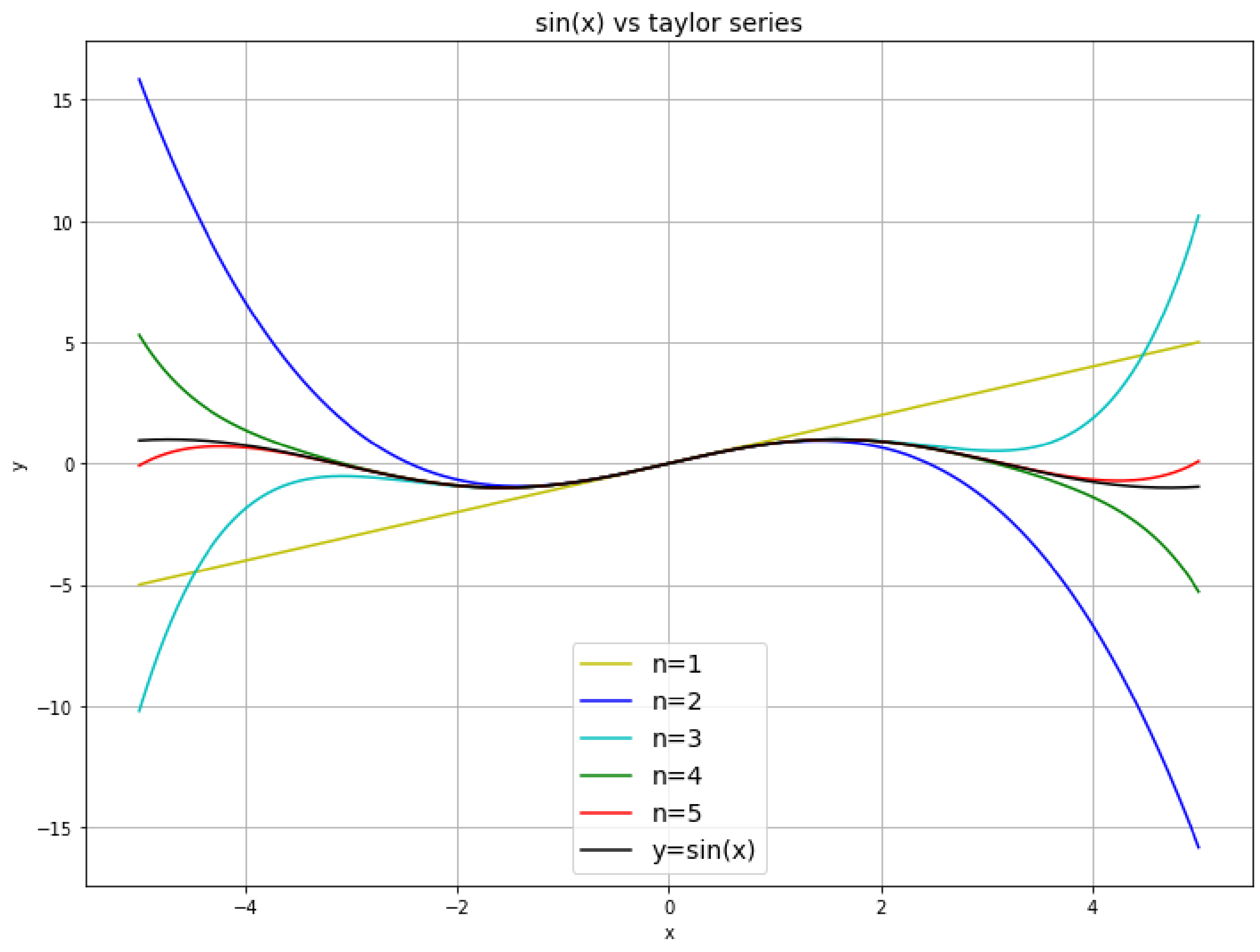
$\exp(x)$ with different number of terms n

```python
def exp_taylor(x, n):
    y = numpy.zeros(x.shape)
    for i in range(n):
        y = y + x**i / factorial(i)  # taylor series for exp(x)
    return y


x = numpy.linspace(0, 5, 501)  # x between 0 and 5

expx = numpy.exp(x)  # actual function

n = [1, 2, 3, 4, 5]  # number of terms between 1 and 5
colour = ['y', 'b', 'c', 'g', 'r']

fig, ax = plt.subplots(1, 2, figsize=(15, 10))

for i in range(len(n)):
    y_exp = exp_taylor(x, n[i])  # values from taylor series
    exp_error = abs(y_exp - expx)  # difference between taylor series and function

    ax[0].plot(x, y_exp, colour[i], label=f'n={n[i]}')
    ax[1].plot(x, exp_error, colour[i], label=f'n={n[i]}')

ax[0].plot(x, expx, 'k', label='y=exp(x)')

ax[0].set_ylabel('y')
ax[0].set_title('exp(x) vs taylor series', fontsize=14)
ax[1].set_ylabel('error')
ax[1].set_title('Error plot of taylor series for exp(x)', fontsize=14)
for i in range(len(ax)):
    ax[i].set_xlabel('x')
    ax[i].legend(loc='best', fontsize=14)
    ax[i].grid(True)

fig.tight_layout()
plt.show()
```

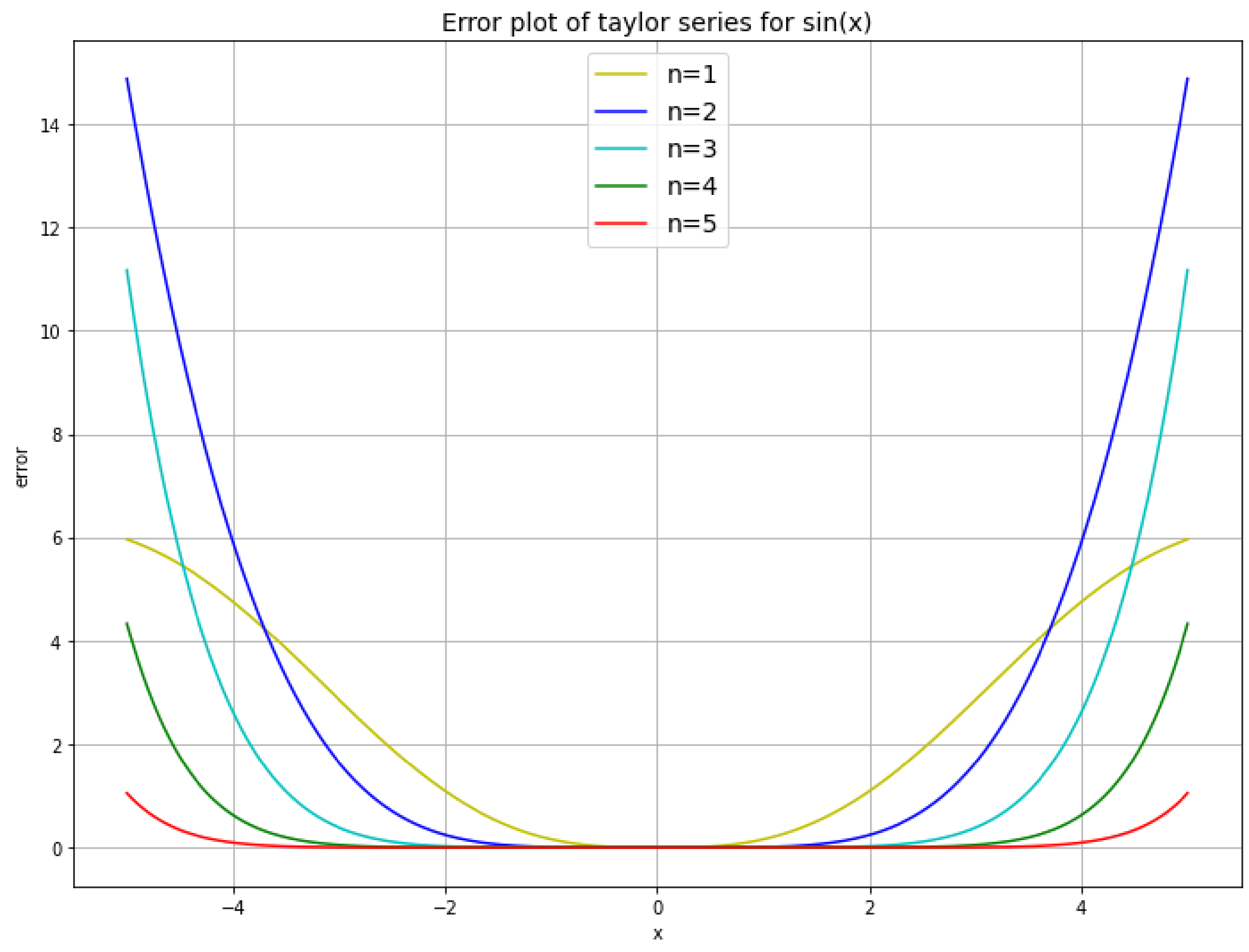


$\sin(x)$ with different number of terms n

```python
def sin_taylor(x, n):
    y = numpy.zeros(x.shape)
    for i in range(n):
        y = y + (-1)**(i)*(x**(2*i+1) / factorial(2*i+1)) # taylor series for sin(x)
    return y

x = numpy.linspace(-5, 5, 1001)  # x between -5 and 5

sinx = numpy.sin(x) # actual function

n = [1, 2, 3, 4, 5]  # number of terms between 1 and 5
colour = ['y', 'b', 'c', 'g', 'r']

fig, ax = plt.subplots(2, 1, figsize=(10, 15))

for i in range(len(n)):
    y_sin = sin_taylor(x, n[i])  # values from taylor series
    sin_error = abs(y_sin - sinx)  # difference between taylor series and function

    ax[0].plot(x, y_sin, colour[i], label=f'n={n[i]}')
    ax[1].plot(x, sin_error, colour[i], label=f'n={n[i]}')

ax[0].plot(x, sinx, 'k', label='y=sin(x)')

ax[0].set_ylabel('y')
ax[0].set_title('sin(x) vs taylor series', fontsize=14)
ax[1].set_ylabel('error')
ax[1].set_title('Error plot of taylor series for sin(x)', fontsize=14)
for i in range(len(ax)):
    ax[i].set_xlabel('x')
    ax[i].legend(loc='best', fontsize=14)
    ax[i].grid(True)

fig.tight_layout()
plt.show()
```



sin(x) vs taylor series
y
x
n=1
n=2
n=3
n=4
n=5
y=sin(x)



Error plot of taylor series for sin(x)
error
x
n=1
n=2
n=3
n=4
n=5

Fourier series

Contents

Mathematics Methods 2 Waves

The Fourier series enables us to represent *periodic* functions as infinite sums. Particularly, it represents functions as a sum of weighted $\sin$ and $\cos$ functions.

This is possible, since the $\sin$ and $\cos$ functions form a complete orthogonal set (*basis functions*). The Fourier series for a function $f(x)$ with a period of 2π is given by:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(nx) + \sum_{n=1}^{\infty} b_n \sin(nx)$$

where a_n and b_n are a set of coeffiecients.

Fourier coeffiecients

Using the orthogonality property of the cos and sin functions we can determine an expression for the coefficients of the Fourier Series. By integrating the Fourier Series over the interval [-π,π], we can determine an expression for a_0. Over this interval, the sums will vanish, as we are intergrating the $\sin$ and $\cos$ functions over one period. Thus, we are left with

$$\int_{-\pi}^{\pi} f(x)dx = \int_{-\pi}^{\pi} \frac{a_0}{2} dx.$$

Therefore, by carrying out the integral we find the expression for a_0 to be

$$a_0 = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x)dx.$$

From this, it is clear that the $\frac{a_0}{2}$ term in the series represents the mean of the function $f(x)$ over the period. In order to determine the a_n coefficients we must multiply the series by $\cos(mx)$, for a positive integer m, and integrate over its period. The series becomes

$$\int_{-\pi}^{\pi} f(x)\cos(mx)dx = \frac{a_0}{2}\int_{-\pi}^{\pi} \cos(mx)dx + \sum_{n=1}^{\infty}\int_{-\pi}^{\pi} a_n \cos(nx)\cos(mx)dx + \sum_{n=1}^{\infty}\int_{-\pi}^{\pi} b_n \sin($$

Due to the orthogonality property of the $\sin$ and $\cos$ functions, the only term that does not vanish on the right-hand side is $\int_{-\pi}^{\pi} a_n \cos(nx)\cos(mx)dx$ for $m = n$. Solving for a_n we get

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos(nx)dx \quad n = 1, 2, \ldots$$

This is consistent with the expresion for a_0 for $n = 0$. Thus the 1/2 factor included on the a_0 term is there to maintain this consistency between a_0 and a_n.

To determine the values of the b_n coefficients we multiply by $\sin(mx)$. Similarly to the a_n coefficients, due to the orthogonality property, the only term that does not vanish on the right-hand side is $\int_{-\pi}^{\pi} b_n \sin(nx)\sin(mx)dx$ for $m = n$. Thus, solving for b_n

$$b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\sin(nx)dx \quad n = 1, 2, \ldots$$

Square wave and Gibbs Phenomenon

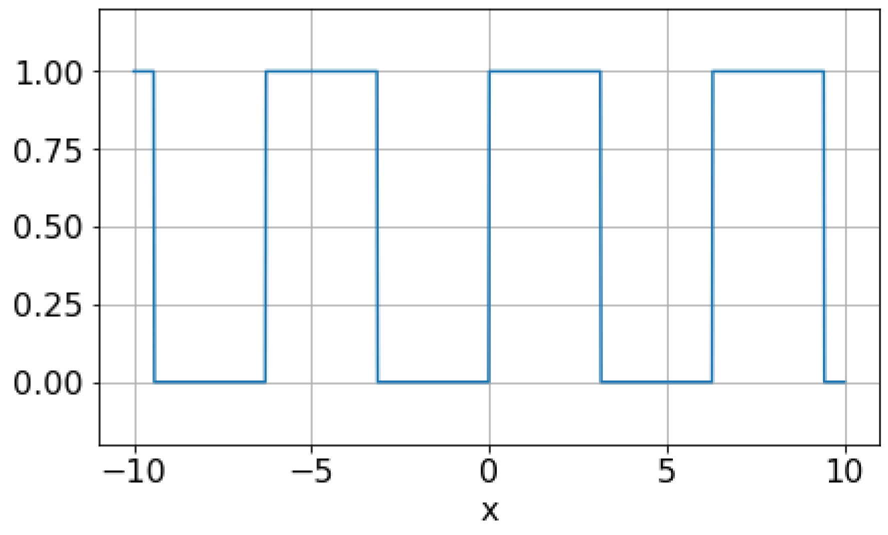
Now that we have introduced the Fourier Series lets look at an example, the positive square wave. This is defined as

$$f(x) = \begin{cases} 0 & \text{if } \; -\pi \le x < 0 \\ 1 & \text{if } \; 0 < x < \pi, \end{cases}$$

Let's plot this function.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
params = {'font.size' : 16 }
matplotlib.rcParams.update(params)
```

```python
# Define a function to create a square wave
def square(x, period):

    # Create array with zeros
    y = np.zeros(len(x))

    # Change zeros to 1 based on a given period
    # in radians (e.g. 2pi, 3pi)
    for i in range(len(x)):
        if (x[i]/(period)) % 1 < 0.50:
            y[i] = 1.0
    return y

N = 1000 # Number of points
x = np.linspace(-10.0, 10.0, N)

plt.figure(figsize=(7,4))
plt.plot(x, square(x, 2*np.pi))
plt.grid(True)
plt.ylim(-0.2, 1.2)
plt.xlabel('x')
plt.show()
```



Now let's try to express the square wave as a Fourier Series. Firstly we need to calculate the coefficients using the above expressions. Carrying out the calculations we find

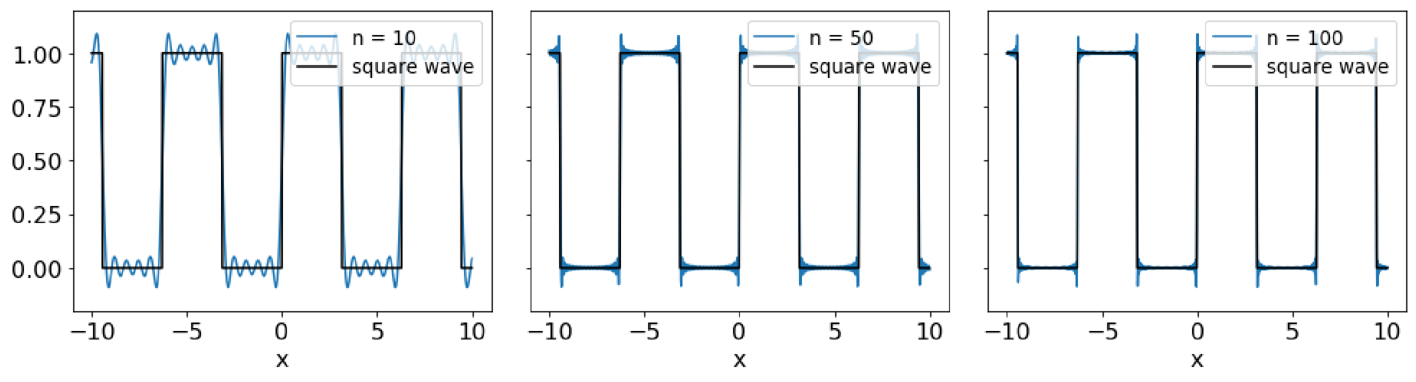
$$a_0 = \frac{1}{\pi}\int_0^{\pi} dx = 1,$$
$$a_n = \frac{1}{\pi}\int_0^{\pi} \cos(nx)dx = 0, \quad n \ge 1,$$
$$b_n = \frac{1}{\pi}\int_0^{\pi} \sin(nx)dx = \begin{cases} 2n\pi & \text{if n is odd,} \\ 0 & \text{if n is even.} \end{cases}$$

Now let's formulate the Fourier Series for the positive square wave. First let's define the Fourier Series and its terms:

```
# Define the cos and sin terms of the Fourier series
def cosTerm(n):
    # Always zero except for n=0
    if n==0: return 1.0
    return 0.

def sinTerm(n):

    if n%2: # n modulo 2 = 1 (True) then Odd
        ret = 2. / (n* np.pi)
    else:
        ret = 0.
    return ret

def fourier(n,x):
    #a_0 term, remember 1/2
    sum = cosTerm(0)/2.0 * np.ones(len(x))

    #all other terms
    for i in range(1, n+1):
        sum += sinTerm(i)*np.sin(i*x) + cosTerm(i)*np.cos(i*x)
    return sum
```

Now let's plot the Fourier Series for $n = 10, n = 50$, and $n = 100$.

```
fig, axes = plt.subplots(1, 3, figsize=(14,4), sharey=True)

# Loop over each subplot
for idx, i in enumerate([10, 50, 100]):
    ax = axes[idx]
    ax.plot(x, fourier(i, x), label="n = %g" % (i))
    ax.plot(x, square(x, 2*np.pi), color="black", label="square wave")
    ax.legend(loc="upper right", fontsize="small")
    ax.set_xlabel("x")
    ax.set_ylim(-0.2, 1.2)

plt.tight_layout()
plt.show()
```



As expected, increasing the number of terms, n, results in the Fourier Series approaching closer to the true shape of the square wave. However, an overshoot (undershoot) and a ringing effect is evident on the plots. This occurs when a truncated Fourier Series is used to approximate a function with discontinuities, such as the function above. In fact, increasing the number of terms will not decrease the amplitude of the overshoot. The effect is present for all functions with discontinuities and is known as the *Gibbs phenomenon*.

Why do the Gibbs phenomena occur? A discontinuity represents a change that takes place in zero time and thus contains infinite high frequencies. By approximating this with a truncated Fourier Series we are including only finite frequencies, which is not enough to properly describe the discontinuity, resulting in the ringing effect around the discontinuity, where the first ring has the largest amplitude.

Fourier series of odd and even functions

Two classes of functions that will be of great use when dealing with Fourier Series are odd and even functions. In this section we will demonstrate how using the properties of odd and even functions can simplify the Fourier Series of a function.

For a symmetric interval around zero we have

$$f(x) = f(-x) \text{ if } f \text{ is even,}$$
$$f(x) = -f(-x) \text{ if } f \text{ is odd.}$$

From these definitions, it is clear that the even functions are symmetric around the y axis, e.g. $\cos(x)$, while the odd functions are antisymmetric around the y axis, e.g. $\sin(x)$.

Properties of odd and even functions

The following properties are very useful in Fourier analysis:

- The product of two odd functions is an even function: $$o_1(x)o_2(x) = e(x).$$
- The product of two even functions is an even function: $$e_1(x)e_2(x) = e(x).$$
- The product of an even and an odd function is an odd function: $$e_1(x)o_1(x) = o(x).$$
- The integral of an odd function on a *symmetric* interval is zero: $$\int_{-a}^{a} o(x)dx = 0.$$
- The integral of an even function on a *symmetric* interval is twice the integral of the postive part: $$\int_{-a}^{a} e(x)dx = 2\int_0^a e(x)dx.$$

Since $\cos(nx)$ is even and $\sin(nx)$ is odd, depending on the nature of $f(x)$ we may be able to eliminate some terms from the Fourier series. Let's use the above example, the positive square wave. The square wave is an odd function, thus the term $f(x)\cos(nx)$ must be odd. Thus, looking at the expression for a_n and the properties of odd functions, we can see that $a_n = 0$ for all $n > 0$, which agrees with what we have found before. Thus, the properties of odd and even functions can allow us to eliminate one of the infinite sums in the Fourier Series, making them very useful in Fourier analysis.

Changing the interval

A function need not be periodic around the [-π,π] interval, in order to be expressed as a Fourier Series. Through a simple change of variable we can express a periodic function across any interval as a Fourier Series. For a function $f(x)$ with a period $p = 2L$, we need to scale the x-axis by a variable u where,

$$u = \frac{\pi x}{L}.$$

Taking into consideration a new function $g(u)$, where $g(u) = f(x)$, we can write the Fourier Series for $g(u)$ as

$$g(u) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(nu) + \sum_{n=1}^{\infty} b_n \sin(nu).$$

Thus, by changing out variable back to x we can write the Fourier Series of $f(x)$ as

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos\left(\frac{n\pi}{L}x\right) + \sum_{n=1}^{\infty} b_n \sin\left(\frac{n\pi}{L}x\right).$$

Using $\frac{du}{dx} = \frac{\pi}{L}$ we can determine the coefficients a_0, a_n and b_n. These are given by

$$\begin{aligned} a_0 &= \frac{1}{L}\int_{-L}^{L} f(x)dx, \\ a_n &= \frac{1}{L}\int_{-L}^{L} f(x)\cos\left(\frac{n\pi}{L}x\right)dx \quad n = 1, 2, \ldots, \\ b_n &= \frac{1}{L}\int_{-L}^{L} f(x)\sin\left(\frac{n\pi}{L}x\right)dx \quad n = 1, 2, \ldots \end{aligned}$$

Using these expressions we can express periodic functions of any period as Fourier Series.

References

- Material used in this notebook was based on the Fourier Series lecture content of Maths Methods 2 module provided by the Earth Science and Engineering Department and on the "Fourier Transforms" course by professor Carlo Contaldi provided by the Physics Department.

Fourier transforms

Contents

Mathematics for Scientists and Engineers 1

Fourier transforms are mathematical operations which when acted on a function, decomose it to its constituent frequencies. In the Fourier Series course, we have shown that a periodic function can be expressed as an infinite sum of sine and cosine functions. We have also shown that, through scaling laws, we can extend the period of the function to an arbitrary length. If the highest frequency in the Fourier Series is kept the same and we keep extending the period of the function, the sum will become longer and longer. In the limit where the period is expanded to infinity, the sum will become an integral, resulting to the definition of the *Fourier Transform*.

The *Fourier Transform* of a function $f(t)$ to a new function $F(\omega)$ is defined as

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt.$$

Using this definition, $f(t)$ is given by the Inverse Fourier Transform

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{i\omega t}d\omega.$$

Using these 2 expressions we can write

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty}\left[\int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt\right]e^{i\omega t}d\omega.$$

This is known as *Fourier's Integral Theorem*. This proves that *any* function can be represented as an infinite sum (integral) of sine and cosine functions, linking back to the Fourier Series.

Note that this definition of the Fourier Transform is not unique. There are many different conventions for the Fourier Transform, but we will stick with this one for this course.

Notation

We will represent the Fourier Transform operator using the calligraphic symbol $\mathcal{F}[f(t)]$, such that

$$\mathcal{F}[f(t)] = F(\omega).$$

Using this notation, we will represent the *Inverse* Fourier transform operator as $\mathcal{F}^{-1}[F(\omega)]$ such that

$$\mathcal{F}^{-1}[F(\omega)] = f(t).$$

Lets look at an example, the top hat function defined as

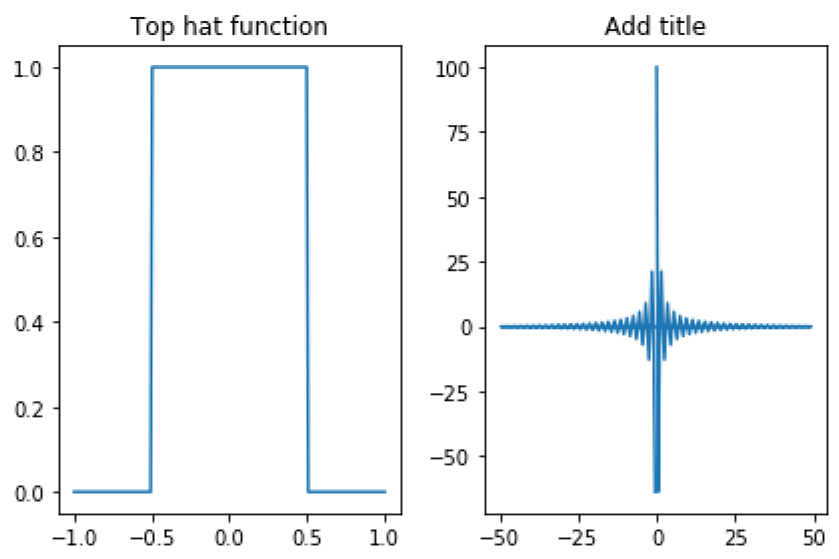
$$\Pi_a(t) = \begin{cases} 1/a & -a/2 < t < a/2, \\ 0 & \text{otherwise.} \end{cases}$$

With a Fourier Transform of

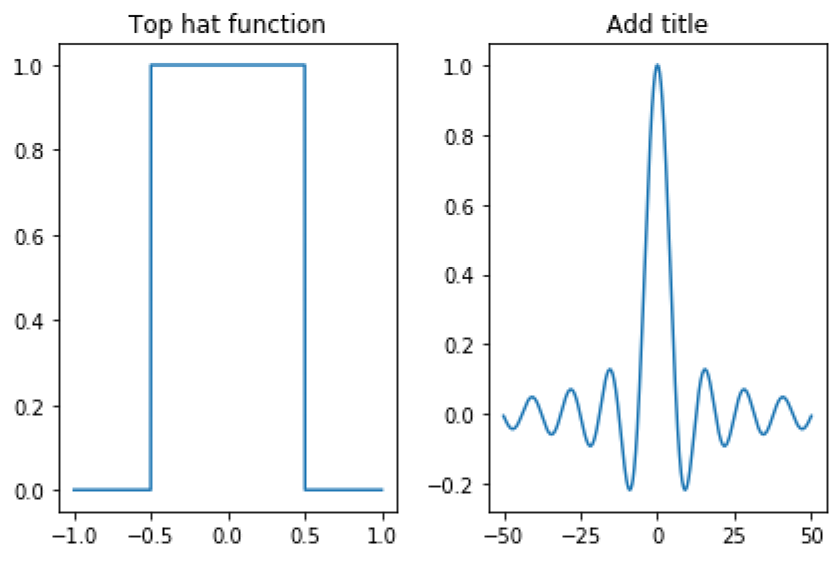
$$F(\omega) = \text{sinc}(\omega/2),$$

let's plot the function and its Fourier Transform.

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the top hat function for a = 1
def top_hat(t):
    if -0.5 < t < 0.5:
        z = 1
    else:
        z=0
    return z

t = np.linspace(-1, 1, 200)
f_t = []
for i in t:
    y = top_hat(i)
    f_t.append(y)

omega = np.fft.fftfreq(len(t),d=t[1] - t[0])
F_omega = np.fft.fft(f_t)

plt.subplot(1,2,1)
plt.plot(t, f_t)
plt.title("Top hat function")

plt.subplot(1,2,2)
plt.title("Add title")
plt.plot(omega, np.real(F_omega), '-')
plt.tight_layout()
plt.show()
```



```python
import numpy as np
import matplotlib.pyplot as plt

# Define the top hat function for a = 1
def top_hat(t):
    if -0.5 < t < 0.5:
        z = 1
    else:
        z=0
    return z

t = np.linspace(-1, 1, 1000)
omega = np.linspace(-50, 50, 1000)

f_t = []
for i in t:
    y = top_hat(i)
    f_t.append(y)
F_omega = []
for i in omega:
    z = np.sinc(i/(2 * np.pi))
    F_omega.append(z)

plt.subplot(1,2,1)
plt.plot(t, f_t)
plt.title("Top hat function")

plt.subplot(1,2,2)
plt.title("Add title")
plt.plot(omega, F_omega)

plt.tight_layout()
plt.show()
```



Special functions and their Fourier Transforms

This section will focus on useful functions and their Fourier Transforms. We will look at the Delta and Gaussian functions.

Delta function

The delta function, $\delta(t)$, is defined as

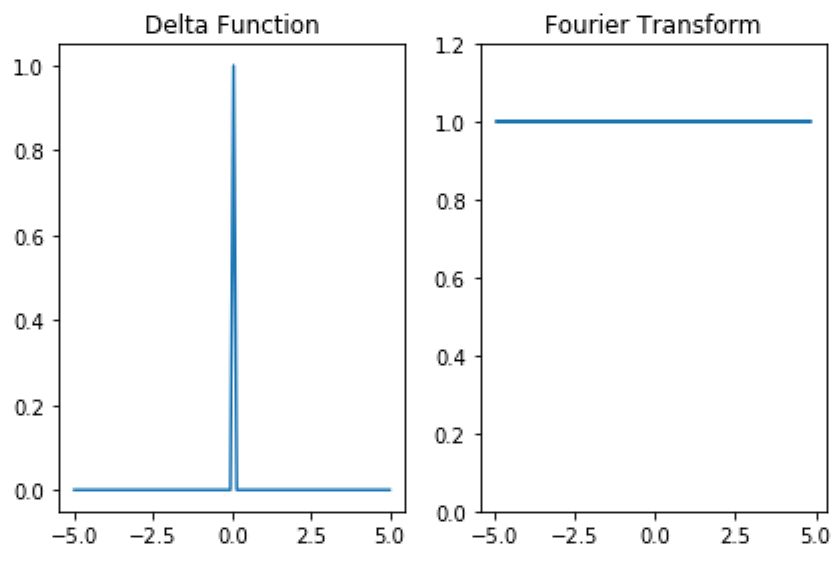
$$\delta(t) = \begin{cases} 0 & t \neq 0, \\ \infty & t = 0. \end{cases}$$

Carrying out the transform we see that Fourier Transform of the delta function is actually a constant, such that

$$\mathcal{F}[\delta(t)] = 1.$$

Let's plot the function and its transform.

```python
from scipy import signal
imp = signal.unit_impulse(100, 'mid') # creates the delta function
t = np.linspace(-5, 5, 100)

plt.subplot(1,2,1)
plt.plot(t, imp)
plt.title('Delta Function')

FT_omega = np.fft.fftfreq(100, t[1] - t[0])
FT = np.fft.fft(imp)

plt.subplot(1,2,2)
plt.plot(FT_omega, abs(FT))
plt.title('Fourier Transform')
plt.ylim([0,1.2])
plt.tight_layout()
plt.show()
```



Gaussian function

A Gaussian function is defined as

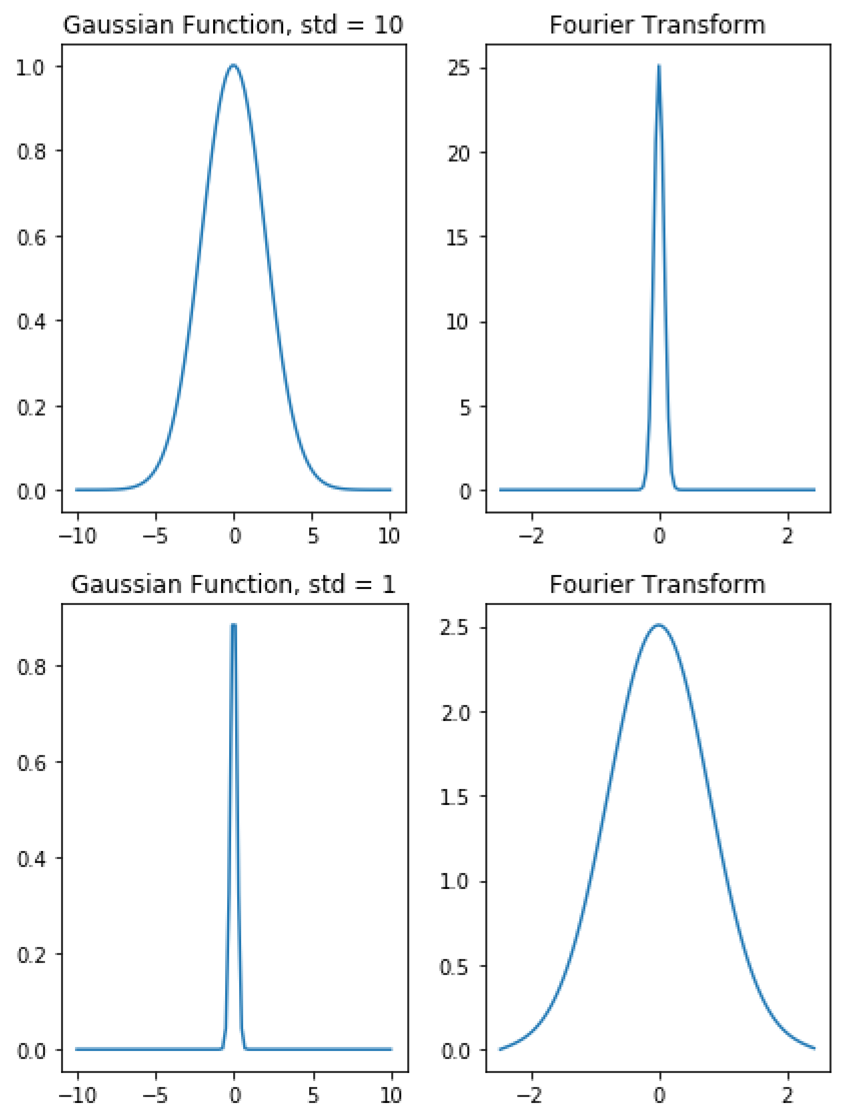
$$f(t) = \exp\left(-\frac{t^2}{2\sigma^2}\right),$$

where σ is the standard devation of the Gaussian. The Fourier transform of a gaussian is another Gaussian such that

$$\mathcal{F}[f(t)] = \sqrt{2\pi}\,\exp\left(-\frac{\omega^2\sigma^2}{2}\right)$$

Thus, we can see that as the Gaussian function gets broader, its Fourier Transform gets narrower. To illustrate this, let's plot the Gaussian and its transform for two different standard deviations.

```
g1 = signal.gaussian(100, std = 10)
t = np.linspace(-10, 10, len(g1))

plt.subplot(1,2,1)
plt.plot(t, g1)
plt.title('Gaussian Function, std = 10')


FT_omega = np.fft.fftfreq(len(g1), t[1] - t[0])
FT = np.fft.fft(g1)
FT_omega = np.fft.fftshift(FT_omega)
FT = np.fft.fftshift(FT)

plt.subplot(1,2,2)
plt.plot(FT_omega, abs((FT)))
plt.title('Fourier Transform')
plt.tight_layout()
plt.show()

g2 = signal.gaussian(100, std = 1)
t = np.linspace(-10,10, len(g2))

plt.subplot(1,2,1)
plt.plot(t, g2)
plt.title('Gaussian Function, std = 1')

FT_omega = np.fft.fftfreq(len(g2), t[1] - t[0])
FT = np.fft.fft(g2)
FT_omega = np.fft.fftshift(FT_omega)
FT = np.fft.fftshift(FT)

plt.subplot(1,2,2)
plt.plot(FT_omega, abs((FT)))
plt.title('Fourier Transform')
plt.tight_layout()
plt.show()
```



Properties of Fourier Transforms

When determining the Fourier Transform of a function, there are a number of properties that can make our calculations easier or even allow us to identify the transform of the function in question as an already known transform. Thus, being familiar with the properties of the Fourier Transforms can be of great use when considering the transforms of specific functions.

Even and odd functions

In general, the Fourier Transform, $F(\omega)$, of a function $f(t)$ will be complex and thus can be written as

$$F(\omega) = R(\omega) + iI(\omega).$$

We can show that the real part of the transform, $R(\omega)$, is related to the even part of the function and that the imaginary part of the transform, $iI(\omega)$, is related to the odd part of the function.

Lets start with an even funtion, $f(t)$, and determine its Fourier Transform.

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt,$$

$$F(\omega) = \int_{-\infty}^{0} f(t)e^{-i\omega t}dt + \int_{0}^{\infty} f(t)e^{-i\omega t}dt,$$

$$F(\omega) = -\int_{0}^{-\infty} f(t)e^{-i\omega t}dt + \int_{0}^{\infty} f(t)e^{-i\omega t}dt,$$

$$F(\omega) = -\int_{0}^{\infty} f(-t)e^{-i\omega t}d(-t) + \int_{0}^{\infty} f(t)e^{-i\omega t}dt.$$

Since $f(t)$ is even, we can use $f(t) = f(-t)$ resulting to

$$F(\omega) = \int_{0}^{\infty} f(t)e^{i\omega t}dt + \int_{0}^{\infty} f(t)e^{-i\omega t}dt,$$

$$F(\omega) = \int_{0}^{\infty} f(t)\left[e^{i\omega t} + e^{-i\omega t}\right]dt,$$

$$F(\omega) = 2\int_{0}^{\infty} f(t)\cos(\omega t)dt.$$

Because $f(t)$ and $\cos(\omega t)$ are both even functions, we can write

$$F(\omega) = \int_{-\infty}^{\infty} f(t)\cos(\omega t)dt.$$

Using a similar procedure, we can derive that for an odd function $f(t)$ the Fourier Transform becomes

$$F(\omega) = -i\int_{-\infty}^{\infty} f(t)\sin(\omega t)dt.$$

Thus, this proves that the Fourier transform of an even function, $e(t)$, is real, while the Fourier Transform of an odd function, $o(t)$, is imaginary. We can take this further, by considering each function, $f(t)$, as a sum of an even and an odd function, such that $f(t) = e(t) + o(t)$. As we stated earlier the Fourier transform of a function can be written as $F(\omega) = R(\omega) + iI(\omega)$. Using these results we can show that

$$R(\omega) = \mathcal{F}[e(t)] = \int_{-\infty}^{\infty} f(t)\cos(\omega t)dt,$$

$$iI(\omega) = \mathcal{F}[o(t)] = -i\int_{-\infty}^{\infty} f(t)\sin(\omega t)dt.$$

Linearity and superposition

A Fourier Transform is linear, meaning that for a function $f(t) = af_1(t) + bf_2(t)$ the Fourier Transform becomes

$$\mathcal{F}\left[f(t)\right] = \mathcal{F}\left[af_1(t) + bf_2(t)\right] = a\mathcal{F}\left[f_1(t)]\right) + b\mathcal{F}\left[f_2(t)\right].$$

This can be easily proven by considering the definition of the Fourier Transform. Again, consider the Fourier Transform of $f(t) = af_1(t) + bf_2(t)$:

$$\mathcal{F}[af_1(t) + bf_2(t)] = \int_{-\infty}^{\infty} [af_1(t) + bf_2(t)]e^{-i\omega t}dt,$$

$$\mathcal{F}[af_1(t) + bf_2(t)] = a\int_{-\infty}^{\infty} [f_1(t)]e^{-i\omega t}dt + b\int_{-\infty}^{\infty} [f_2(t)]e^{-i\omega t}dt,$$

$$\mathcal{F}[af_1(t) + bf_2(t)] = aF_1(\omega) + bF_2(\omega).$$

Reciprocal broadening/scaling

Stretching a function by a factor α, results in the Fourier Transform of the function to be compressed by the same factor. Consider the transform of a function, $f(\alpha t)$

$$\mathcal{F}[f(\alpha t)] = \int_{-\infty}^{\infty} f(\alpha t)e^{-i\omega t}dt,$$

$$\mathcal{F}[f(\alpha t)] = \frac{1}{|\alpha|}\int_{-\infty}^{\infty} f(\alpha t)e^{\frac{-i\omega\alpha t}{\alpha}}d(\alpha t),$$

$$\mathcal{F}[f(\alpha t)] = \frac{1}{|\alpha|}F\left(\frac{\omega}{\alpha}\right).$$

This shows that as the functions gets broader, its transform not only becomes narrower but due to the $1\,/\,|\alpha|$ factor it also increases in amplitude.

Translation

Shifting a function by a certain amount results in a phase shift on the Fourier Transform:

$$\mathcal{F}[f(t - t_0)] = e^{-i\omega t_0}F(\omega),$$

$$\mathcal{F}[f(t - t_0)e^{-it\omega_0}] = F(\omega - \omega_0).$$

Derivatives and integrals

Finding the transform of derivative simply translates to a multiplication of the original transform such that

$$\mathcal{F}[f'(t)] = i\omega F(\omega).$$

To prove this we must differentiate the function, $f(t)$, and take its transform

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{i\omega t}d\omega,$$

$$\text{differentiating, we get}\quad f'(t) = \int_{-\infty}^{\infty} i\omega F(\omega)e^{i\omega t}d\omega.$$

This can be indentified as the inverse Fourier Transform, such that

$$f'(t) = \mathcal{F}^{-1}[i\omega F(\omega)].$$

Thus, taking the Fourier Transform of $f'(t)$ proves the above statement

$$\mathcal{F}[f'(t)] = \mathcal{F}[\mathcal{F}^{-1}[i\omega F(\omega)]],$$

$$\mathcal{F}[f'(t)] = i\omega F(\omega).$$

We can generalise this result to higher order derivatives. For the nth derivative

$$\mathcal{F}[f'^{(n)}(t)] = (i\omega)^n F(\omega).$$

This shows that differentiation magnifies *high frequencies* and shifts the phase of the transform by $\pi\,/\,2$.

Using this result we can show how integration affects the transform of a function

$$\int \mathcal{F}[f(t)]dt = F(\omega)\,/\,i\omega \ + \text{constant.}$$

Convolution

When discussing the nature of Fourier Series and Transforms, one needs to discuss *convolutions*. Simply stated, a convolution of two (or more) functions is defined as the integral over *all space* of the product of the *two* desired functions after one has been *reversed and shifted*.

The convolution of two functions, $a(t)$ and $b(t)$ is denoted by $a(t) * b(t)$ and defined as

$$a(t) * b(t) = \int_{-\infty}^{\infty} a(u)b(t-u)du$$

where u is a dummy variable that dissapears when integrating. It should be noted that convolution is commutative, meaning that the ordering is not important.

Convolution is perhaps one of the most important tools for a scientist of any discipline. This can be illustrated via a simple example. Imagine you have just made measurements of the magnitude of a magnetic field at a particular direction using a magnetometer. Those measurements have an inherent error due to the precision of the instrument you used. This error will lead to the *smearing* of your outcome distribution, or in other words the true distribution has been *convolved* with the error function. Therefore in order to recover the original (true) distrubution of your measurements you need to use the *Convolution Theorem*, detailed below.

The convolution theorem

Let us examine what happens when we apply a Fourier Transform on a convolution of two functions.

$$\mathcal{F}[f_1(t) * f_2(t)] = \int_{-\infty}^{\infty}\left(\int_{-\infty}^{\infty} f_1(u)f_2(t-u)du\right)e^{-i\omega t}dt,$$

$$\mathcal{F}[f_1(t) * f_2(t)] = \int_{-\infty}^{\infty} f_1(u)e^{-i\omega u}\left(\int_{-\infty}^{\infty} f_2(t-u)e^{-i\omega(t-u)}dt\right)du,$$

$$\mathcal{F}[f_1(t) * f_2(t)] = \left(\int_{-\infty}^{\infty} f_1(u)e^{-i\omega u}du\right)\left(\int_{-\infty}^{\infty} f_2(s)e^{-i\omega(s)}ds\right),$$

$$\mathcal{F}[f_1(t) * f_2(t)] = F_1(\omega)F_2(\omega),$$

where the splitting of the integrals comes from making the substitution $s = t - u$ and then noting that u no longer appears on the inner integral. The final expression deduced above is known as *The Convolution Theorem*. It is one of the most important properties of Fourier Transfors which is evident as it the essence of Fourier Analysis. The Convolution Theorem states that the Fourier Transform of the convolution of two functions is equal to the product of the Fourier Transforms of each function. Looking at it in the opposite direction, the Fourier Transform of the product of two functions is given by the convolution of the Fourier Transform of those functions individually:

$$\mathcal{F}[f_1(t)f_2(t)] = \frac{1}{2\pi}F_1(\omega) * F_2(\omega).$$

Going back to the problem we discussed at the beginning of the notebook, we can now utilise the Convolution Theorem to understand how to retrieve the true distribution from a set of data that has been convolved with an error function. Simply, apply a Fourier transform on the resulting convolved distribution and divide it by the Fourier Transform of the known error function. The result will be the Fourier Transform of your true distribution.

TODO: **ADD AN EXAMPLE OF HOW CONVOLUTION CAN BE USED IN CODE?**

References

- Material used in this notebook was based on the “Fourier Transforms” course by professor Carlo Contaldi provided by the Physics Department.

Taylor series

Contents

Mathematics Methods 1 Mathematics Methods 2 Numerical Methods

A Taylor series is a method by which we can approximate an arbitrary function by a sum of terms made up of that function and its derivatives at a single point.

This concept underlies many things (especially in computational science) so it’s worth us spending some time reviewing it.

Motivation - a constant approximation

Consider a function of a single independent variable: $f(x)$

Suppose this function is really really expensive to evaluate. Suppose I have already evaluated it at a single x value, call it x_0, i.e. I know the value of $f(x_0)$.

Given only this information, what is the best guess (estimation/approximation) I can make for $f(x)$ for a choice $x \neq x_0$?

Well I can’t really do anything better than

$$f(x) \approx f(x_0)$$

This may seem a hopeless approximation, but it can be used in some situations. Pretend f is a weather forecast and x is time. This just says that for a forecast of the weather tomorrow use what we see today - this is a real technique called *persistent forecast* (“today equals tomorrow”).

”… This makes persistence a ‘hard to beat’ method for forecasting longer time periods”: http://ww2010.atmos.uiuc.edu/(Gh)/guides/mtr/fcst/mth/prst.rxml

Let’s consider an example:

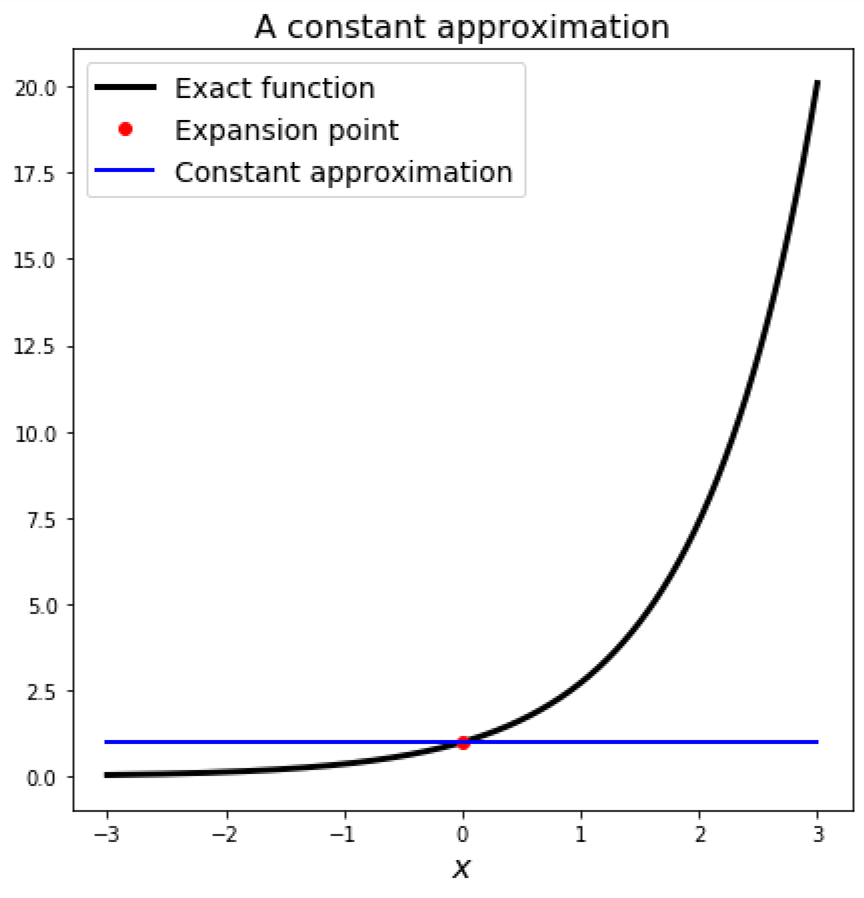
```python
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

```python
def f(x):
    return np.exp(x)
```

and plot our current situation, assuming that $x_0 = 0$. We call this the expansion point - we are constructing an approximation “around this point”.

```python
fig = plt.figure(figsize=(7, 7))
ax1 = plt.subplot(111)

ax1.set_title('A constant approximation', fontsize=16)
ax1.set_xlabel('$x$', fontsize=16)

# define our x for plotting purposes
x = np.linspace(-3., 3., 1000)

# plot exact function
ax1.plot(x, f(x), 'k', lw=3, label='Exact function')

# define our approximation
x0 = 0.0
y = f(x0)*np.ones_like(x)
ax1.plot(x0, f(x0), 'ro', label='Expansion point')

# and plot the constant approximation
ax1.plot(x, y, 'b', lw=2, label='Constant approximation')

ax1.legend(loc='best', fontsize=14)
```

```
<matplotlib.legend.Legend at 0x2322df4cb38>
```



Can I have any confidence in this guess?

1. Yes if it turns out that the function f is actually a constant.
2. Yes, if I can assume that f doesn't vary very much in the vicinity of x_0, and if we are interested in $x$ values that are close to x_0.

How can I do better?

Motivation - a linear approximation (a "linearisation")

The obvious improvement we can make is to approximate the true function with a linear approximation rather than a constant.

This is a common way to write a linear function:

$$y = mx + c$$

where m is the slope and c is the intercept - the y value the line hits the $x = 0$ axis at.

We consider an approximation, or an expansion, (or in this case a "linearisation") "about a point".

For us the point is x_0 and it makes sense for us to make the slope of the linear approximation the same as the derivative of the function at this point.

So we are now using the value of f at x_0 AND the value of the derivative of f at x_0: $m = f'(x_0)$.

This is the formula for the linear line with this slope:

$$y = f'(x_0)x + c$$

where choice of c moves the linear line up and down. We want it to pass through the function at x_0 as well of course, and this allows us to figure out what value c should take.

Actually we generally write the linearisation in the following form

$$f(x) \approx f(x_0) + (x - x_0)f'(x_0)$$

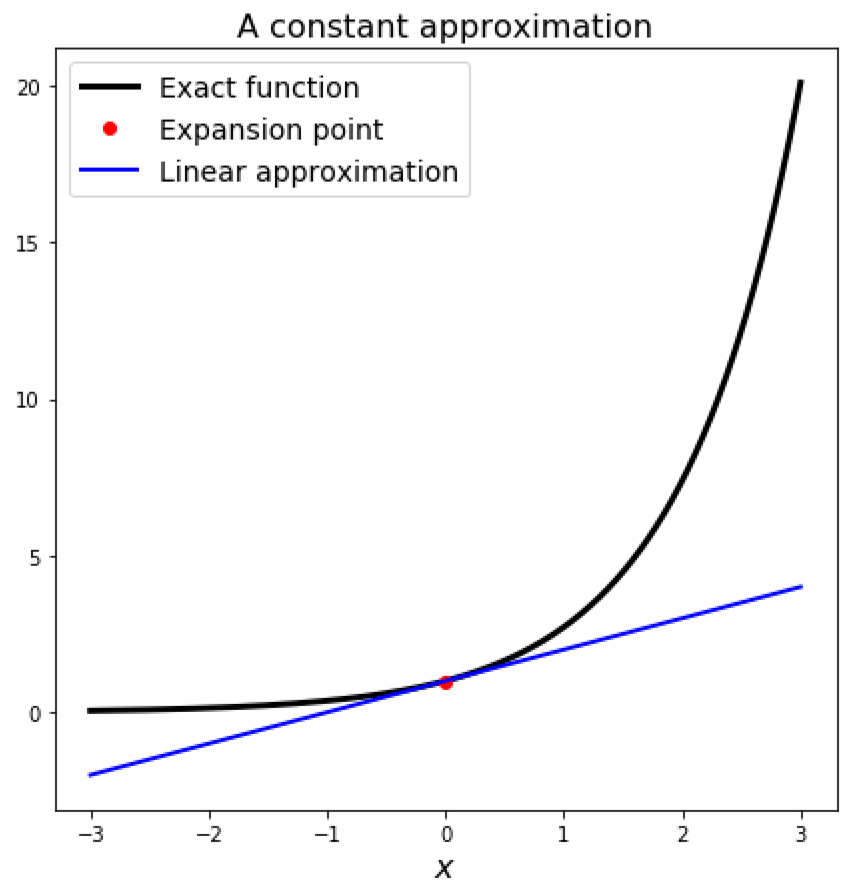
so when $(x - x_0)$ is zero, we are at the expansion point and we indeed get $f(x_0)$ on the RHS. As we move away from that point $(x - x_0)$ grows and our approximation adds a correction based on the size of the derivative at x_0.

Let's plot this. First we need to define the derivative:

```
# of course for our example f this is trivial:
def fx(x):
    return np.exp(x)
```

```
fig = plt.figure(figsize=(7, 7))
ax1 = plt.subplot(111)

ax1.set_title('A constant approximation', fontsize=16)
ax1.set_xlabel('$x$', fontsize=16)

# define our x for plotting purposes
x = np.linspace(-3., 3., 1000)

# plot exact function
ax1.plot(x, f(x), 'k', lw=3, label='Exact function')

# define our approximation
x0 = 0.0
y = (f(x0) + (x-x0)*fx(x0))*np.ones_like(x)
ax1.plot(x0, f(x0), 'ro', label='Expansion point')

# and plot the constant approximation
ax1.plot(x, y, 'b', lw=2, label='Linear approximation')

ax1.legend(loc='best', fontsize=14)
```

```
<matplotlib.legend.Legend at 0x2322e095cf8>
```



This is clearly an improved approximation compared to the constant one.

However the same accuracy points hold: if we are a long way from the expansion point, or if f is very complex, this may not be a good approximation.

Can we do better - yes we can consider the curvature of the black line in addition to its value and its slope, i.e. we can include a term proportional to $f''(x_0)$ to our approximation.

and so on...

This is the formula for the infinite Taylor series *about (or around) the point* x_0

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2!}f''(x_0) + \frac{(x - x_0)^3}{3!}f'''(x_0) + \frac{(x - x_0)^4}{4!}f^{(iv)}(x_0) +$$

An equivalent way of writing this expansion is

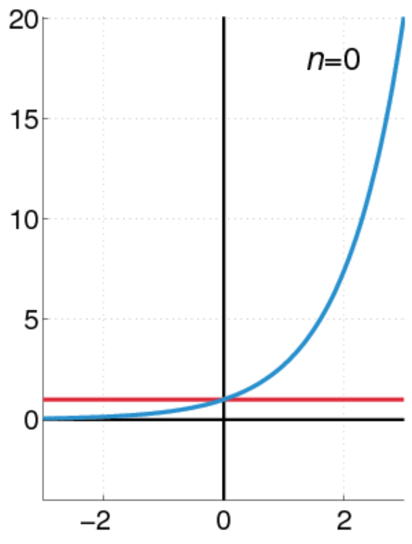
$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2!}f''(x_0) + \frac{h^3}{3!}f'''(x_0) + \dots$$

or replace h by the notation Δx or δx.

As we include more and more terms our approximation improves - see the following animated gif from Wikipedia which explains the point.

Taylor series example

Wikipedia image: The exponential function (in blue), and the sum of the first (n + 1) terms of its Taylor series expansion around the point 0 (in red).



More terms equate with a better approximation valid a larger distance from x_0.

Aside: Big O notation

When talking about terms in infinite series (terms we will often be forces to truncate, i.e. throw away), or talking about errors, convergence, complexity, run times etc, so-called Big-O notation is very useful.

As an alternative to writing "$\dots$" in the above infinite expansions, we can also write

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2!}f''(x_0) + \frac{(x - x_0)^3}{3!}f'''(x_0) + \mathcal{O}((x - x_0)^4)$$

or

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2!}f''(x_0) + \frac{h^3}{3!}f'''(x_0) + \mathcal{O}(h^4)$$

What does this mean?

LINK TO THE NOTEBOOK ON BIG O NOTATION

Truncation error

In principle we can use Taylor series to approximate a (sufficiently smooth) function with arbitrary accuracy as long as we use sufficiently many terms, but in practice we will have to truncate.

In the next section we will use similar ideas to compute an approximate solution to a differential equations, we will do this by approximating the operation of taking a derivative.

Again by making use of enough terms from a Taylor series we can construct approximations of derivatives with arbitrary accuracy, but again in practice we will have to truncate.

In both cases the act of limiting the number of terms we use introduces an error.

Since we are truncating an infinite series at some point, this type of error is often called a *truncation error*.

FURTHER READING:

For an application of Taylor series see the notebook on timestepping ODEs