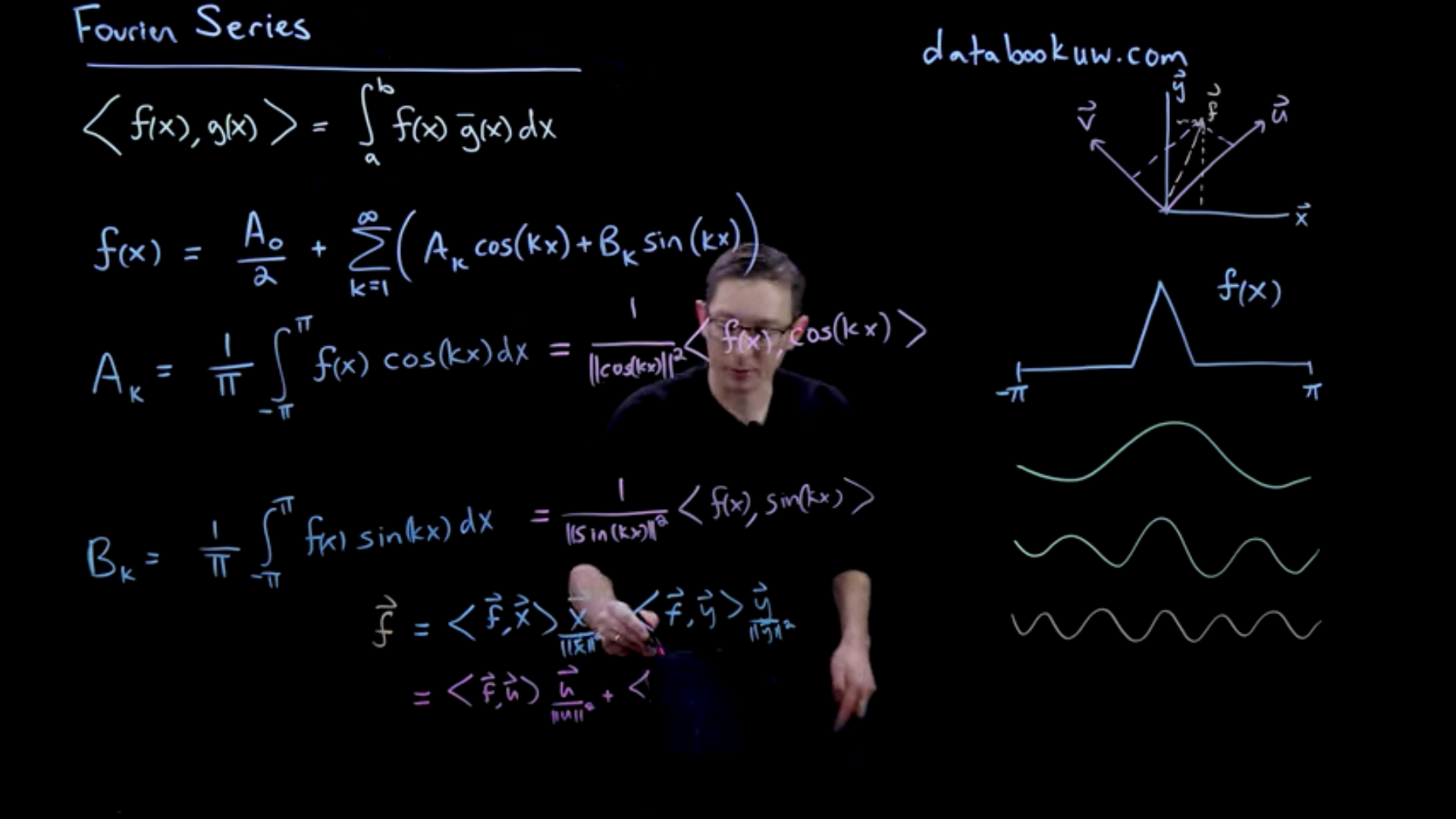
DAILY ASSESSMENT FORMAT

|  |  |  |  |
| --- | --- | --- | --- |
| Date | 25/05/2020 | Name: | Prajna |
| Course: | Digital signal processing | USN: | 4AL16EC047 |
| Topic: | Introduction to Fourier Series & Fourier Transform | Semester &  Section: | 8 “A” |
| FORENOON SESSION DETAILS | | | |

**Report**

**Introduction to Fourier Series & Fourier Transform**

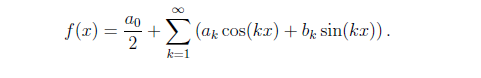


Fast forward two hundred years, and the fast Fourier transform has become the cornerstone of computational mathematics, enabling real-time image and audio compression, global communication networks, modern devices and hardware, numerical physics and engineering at scale, and advanced data analysis.

Simply put, the fast Fourier transform has had a more significant and profound role in shaping the modern world than any other algorithm to date. With increasingly complex problems, data sets, and computational geometries, simple Fourier sine and cosine bases have given way to tailored bases, such as the data-driven SVD. In fact, the SVD basis can be used as a direct analogue of the Fourier basis for solving PDEs with complex geometries. In addition, related functions, called wavelets, have been developed for advanced signal processing and compression efforts.

**Fourier series**

A fundamental result in Fourier analysis is that if f(x) is periodic and piece wise smooth, then it can be written in terms of a Fourier series, which is an infinite sum of cosines and sine of increasing frequency. In particular, if f(x) is 2\_-periodic, it may be written as:



Fourier series approximation to a hat function

% define domain

dx = 0.001;

L = pi;

x = (-1+dx:dx:1)\*L;

n = length(x); nquart = floor(n/4);

% define hat function

f = 0\*x;

f(nquart:2\*nquart) = 4\*(1:nquart+1)/n;

f(2\*nquart+1:3\*nquart) = 1-4\*(0:nquart-1)/n;

plot(x,f,’-k’,’LineWidth’,1.5), hold on

% Compute Fourier series

CC = jet(20);

A0 = sum(f.\*ones(size(x)))\*dx;

fFS = A0/2;

for k=1:20

A(k) = sum(f.\*cos(pi\*k\*x/L))\*dx; % Inner product

B(k) = sum(f.\*sin(pi\*k\*x/L))\*dx;

fFS = fFS + A(k)\*cos(k\*pi\*x/L) + B(k)\*sin(k\*pi\*x/L);

plot(x,fFS,’-’,’Color’,CC(k,:),’LineWidth’,1.2)

end

**Fourier series for a discontinuous hat function**

dx = 0.01; L = 10;

x = 0:dx:L;

n = length(x); nquart = floor(n/4);

f = zeros(size(x));

f(nquart:3\*nquart) = 1;

A0 = sum(f.\*ones(size(x)))\*dx\*2/L;

fFS = A0/2;

for k=1:100

Ak = sum(f.\*cos(2\*pi\*k\*x/L))\*dx\*2/L;

Bk = sum(f.\*sin(2\*pi\*k\*x/L))\*dx\*2/L;

fFS = fFS + Ak\*cos(2\*k\*pi\*x/L) + Bk\*sin(2\*k\*pi\*x/L);

end

plot(x,f,’k’,’LineWidth’,2), hold on

plot(x,fFS,’r-’,’LineWidth’,1.2)

**Fourier series using python**

# #### test that it works with real coefficients:

from numpy import linspace, allclose, cos, sin, ones\_like, exp, pi, \

complex64, zeros

def series\_real\_coeff(a0, a, b, t, T):

"""calculates the Fourier series with period T at times t,

from the real coeff. a0,a,b"""

tmp = ones\_like(t) \* a0 / 2.

for k, (ak, bk) in enumerate(zip(a, b)):

fFS = fFS + A(k)\*cos(k\*pi\*x/L) + B(k)\*sin(k\*pi\*x/L);

plot(x,fFS,’-’,’Color’,CC(k,:),’LineWidth’,1.2)

end

**Fourier series for a discontinuous hat function**

dx = 0.01; L = 10;

x = 0:dx:L;

n = length(x); nquart = floor(n/4);

f = zeros(size(x));

f(nquart:3\*nquart) = 1;

A0 = sum(f.\*ones(size(x)))\*dx\*2/L;

fFS = A0/2;

for k=1:100

Ak = sum(f.\*cos(2\*pi\*k\*x/L))\*dx\*2/L;

Bk = sum(f.\*sin(2\*pi\*k\*x/L))\*dx\*2/L;

fFS = fFS + Ak\*cos(2\*k\*pi\*x/L) + Bk\*sin(2\*k\*pi\*x/L);

end

plot(x,f,’k’,’LineWidth’,2), hold on

plot(x,fFS,’r-’,’LineWidth’,1.2)

**Fourier series using python**

# #### test that it works with real coefficients:

from numpy import linspace, allclose, cos, sin, ones\_like, exp, pi, \

complex64, zeros

def series\_real\_coeff(a0, a, b, t, T):

"""calculates the Fourier series with period T at times t,

from the real coeff. a0,a,b"""

tmp = ones\_like(t) \* a0 / 2.

for k, (ak, bk) in enumerate(zip(a, b)):

tmp += ak \* cos(2 \* pi \* (k + 1) \* t / T) + bk \* sin(

2 \* pi \* (k + 1) \* t / T)

return tmp

t = linspace(0, T, 100)

f\_values = f(t)

a0, a, b = fourier\_series\_coeff\_numpy(f, T, 52)

# construct the series:

f\_series\_values = series\_real\_coeff(a0, a, b, t, T)

# check that the series and the original function match to numerical precision:

assert allclose(f\_series\_values, f\_values, atol=1e-6)

# #### test similarly that it works with complex coefficients:

def series\_complex\_coeff(c, t, T):

"""calculates the Fourier series with period T at times t,

from the complex coeff. c"""

tmp = zeros((t.size), dtype=complex64)

for k, ck in enumerate(c):

# sum from 0 to +N

tmp += ck \* exp(2j \* pi \* k \* t / T)

# sum from -N to -1

if k != 0:

tmp += ck.conjugate() \* exp(-2j \* pi \* k \* t / T)

return tmp.real

f\_values = f(t)

c = fourier\_series\_coeff\_numpy(f, T, 7, return\_complex=True)

f\_series\_values = series\_complex\_coeff(c, t, T)

assert allclose(f\_series\_values, f\_values, atol=1e-6)

|  |  |  |  |
| --- | --- | --- | --- |
| Date | 25/05/2020 | Name: | Prajna |
| Course: | PYTHON | USN: | 4AL16EC047 |
| Topic: | Functions | Semester &  Section: | 8 “A” |
| AFTERNOON SESSION DETAILS | | | |

**Functions Basics and little Advanced:**

A **function** is a set of statements that take inputs, do some specific computation and produces output.  **Python** provides built-in **functions** like print(), etc. but we can also create your own **functions**. These **functions** are called user-defined **functions**.

* Function takes input and spits off an output.
* Inputs are called argument, parameters, inputs, passing values/reference.
* These functions dont return anything.
* After you write a simple code you convert it into functions.

def print\_two(arg1, arg2):

print(f"arg1: {arg1}, arg2: {arg2}")

# this just takes one argument

def print\_one(arg1):

print(f"arg1: {arg1}")

# this one takes no arguments

def print\_none():

print("I got nothin' to return.")

print\_two("S", "J")

print\_one("S!")

print\_none()

Below are the functions that would return the calculation.Create errors and give only one or three arguments.

def add(a, b):

print(a,b)

return a + b

def subtract(a, b):

print(a,"minus",b)

return a - b

def multiply(a, b):

print("multiple",a,b)

return a \* b

def divide(a, b):

print("DIVIDING",a,b)

return a / b

print("Let's see some functions!")

age = add(30, 5)

height = subtract(78, 4)

weight = multiply(90, 2)

iq = divide(100 , 2)

Try to see if you can find out the order of execution for the above:

def xxx(\*args):

for xy in args:

print(xy)

xxx('aaa','bbb','ccc','eee')

This taken in multiple arguments.

x = "I like the way python works"

y = x.split()

def print\_first\_word(words):

"""Prints the first word after popping it off."""

word = words.pop(0)

print(word)

print\_first\_word(y)

y

First recursive function!

What the is recursion?

Recursion is a way of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition fulfils the condition of recursion, we call this function as recursive function.

def factorial( n ):

if n <1: # base case

return 1

else:

returnNumber = n \* factorial( n - 1 ) # recursive call

print(str(n) + '! = ' + str(returnNumber))

return returnNumber

def foo(\*args):

for a in args:

print (a)

foo(1,2,3,5,6)

Lambda function:

In python, a Lambda function is a single line function declared with no name, which can have any one arguments, but it can only have one expression.Such a function is capable of behaving similarly to a regular function declared using the Python’s def keyword.

* create functions on the fly in python in minimal code.
* JS allows you to the same. and now scala.

def short\_function(x):

return x \* 2

f = lambda x: x \* 2

f(6)

short\_function(6)

Passing function as a argument:

def apply\_to\_list(some\_list, short\_function):

return [short\_function(x) for x in some\_list]

apply\_to\_list([1,2,3],short\_function )

ints = [4, 0, 1, 5, 6]

apply\_to\_list(ints, lambda x: x \* 2)

strings = ['foo', 'card', 'bar', 'aaaa', 'abab']

strings.sort(key=lambda x: len(set(list(x))))

strings

Try to convert, if you cannot convert return back what you have.

def attempt\_float(x):

try:

return float(x)

except:

return x

Difference between printing and returning:

def joshif(x, y, z):

print("function ran - message from inside of function")

print ("end of the function")

return (x + y) \* z

Optional argument with if else:

def add\_and\_maybe\_multiply(a, b, c=None):

result = a + b

if c is not None:

result = result \* c

return result