Complex Analysis Using Python

Submitted By:
Pranav Gopalkrishna

Register Number:
1940223

CHRIST (DEEMED TO BE UNIVERSITY)

DEPARTMENT OF MATHEMATICS

BSC CMS, 6TH SEMESTER

September 08, 2021

Contents

1 Introduction

In this course, i.e. Complex Analysis using Python programming, we learn to use Python programming concepts and libraries to apply and verify Complex Analysis concepts that we have learnt in our theory classes.

2 Introduction to CMath

AIM: CMath module for dealing with complex numbers.

[11]:
```
import cmath as cm
dir(cm)
```

[11]:
```
['__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atanh',
 'cos',
 'cosh',
 'e',
 'exp',
 'inf',
 'infj',
 'isclose',
 'isfinite',
 'isinf',
 'isnan',
 'log',
 'log10',
 'nan',
 'nanj',
 'phase',
 'pi',
 'polar',
 'rect',
```

```
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
 'tau']
```

Note that the 'a' in acos, asin, atan, etc. means 'arc' i.e. acos is arccosine, asin is arcsine, etc.

2.1 Read complex number & apply some CMath functions

[13]:
```python
# A valid complex number string must be in the form a+bj
# a and b must be numeric values, and j represents sqrt(-1)
# No spaces must be between the complex number characters
# Spaces may be put before and after the complex number
z = complex(input(">> "))
```

```
>> 3+8j
```

[29]:
```python
print("Sine:", cm.sin(z))
print("Cosine:", cm.cos(z))
print("Square root: ", cm.sqrt(z))
```

```
Sine: (210.3364312489715-1475.5628538734973j)
Cosine: (-1475.5631859789817-210.33638390848276j)
Square root:  (2.402499089002692+1.6649329934441104j)
```

2.2 Verifying commutativity & associativity

(of addition and multiplication of complex numbers)

[34]:
```python
print("Input three complex numbers:")
z1 = complex(input(">> "))
z2 = complex(input(">> "))
z3 = complex(input(">> "))
```

```
Input three complex numbers:
>> 3-7j
>> 2-9j
>> 3+6j
```

[35]:
```python
print("Commutative w.r.t. addition?", z1+z2 == z2+z1)
print("Commutative w.r.t. multiplication?", z1*z2 == z2*z1)
```

```
Commutative w.r.t. addition? True
Commutative w.r.t. multiplication? True
```

[38]:
```
print("Associative w.r.t. addition?", z1+(z2+z3) == (z1+z2)+z3)
print("Associative w.r.t. multiplication?", z1*(z2*z3) == (z1*z2)*z3)
```

```
Associative w.r.t. addition? True
Associative w.r.t. multiplication? True
```

2.3 Magnitude of complex number

[27]:
```
z = complex(4, -8)
print("Complex number:", z)
#------------
print("Magnitude (using 'abs' function):", abs(z))
from math import sqrt
print("Magnitude (using formula):", sqrt(z.real**2 + z.imag**2))
```

```
Complex number: (4-8j)
Magnitude (using 'abs' function): 8.94427190999916
Magnitude (using formula): 8.94427190999916
```

2.4 Phase of a complex number

Phase of a complex number is simply the argument of the complex number, i.e. the angle between the vector representing the complex number in the complex plane and the real number axis in the same plane. For a complex number x + iy, the phase i.e. argument is given by $tan^{-1}(\frac{y}{x})$ i.e. $tan^{-1}(\frac{im(z)}{re(z)})$.

[28]:
```
z = complex(3, 4)
print("Complex number:", z)
#------------
print("Phase (using 'phase' function):", cm.phase(z))
from numpy import arctan
# (Using purely numeric arctan function for demo purposes, though CMath␣
↪also offerns arctan).
print("Phase (using arctan(im(z)/re(z))):", arctan(z.imag/z.real))
```

```
Complex number: (3+4j)
Phase (using 'phase' function): 0.9272952180016122
Phase (using arctan(im(z)/re(z))): 0.9272952180016122
```

2.5 Polar coordinates of a complex number

2.5.1 Representing in polar form

Polar coordinates represent a complex number using the complex number's magnitude and argument (i.e. phase). For a complex number z, if the magnitude is r and the argument is θ, then the polar coordinates of z are (r, θ).

[30]:
```
z = complex(1, 7)
print("Complex number:", z)
#------------
print("Polar coordinates (using 'abs' and 'phase'):", (abs(z), cm.
 ↪phase(z)))
print("Polar coordinates (using 'polar' function):", cm.polar(z))
```

```
Complex number: (1+7j)
Polar coordinates (using 'abs' and 'phase'): (7.0710678118654755,
1.4288992721907328)
Polar coordinates (using 'polar' function): (7.0710678118654755,
1.4288992721907328)
```

2.5.2 Polar coordinates to rectangular coordinates

For a complex number $z = x + iy$, its rectangular coordinates are (x, y). Rectangular coordinates are simply the Cartesian representation of the complex number, by giving the real and imaginary parts as the x and y coordinates respectively.

[47]:
```
p = (2, 4)
# For this complex number, magnitude is 2 and argument is 4.
print("Complex number (polar coordinates):", z)
#------------
print("Rectangular coordinates (using 'rect'):", cm.rect(p[0], p[1]))
```

```
Complex number (polar coordinates): (1+7j)
Rectangular coordinates (using 'rect'):
(-1.3072872417272239-1.5136049906158564j)
```

2.6 Conjugate

Conjugate of a complex number can be found by simply inverting the sign of the imaginary part. In Python, we have an in-built attribute method for complex numbers '.conjugate( )' that evaluates the conjugate of a given complex number.

[1]:
```
z = complex(1, 7)
print("Complex number:", z)
```

```
print("Conjugate:", z.conjugate())
```

```
Complex number: (1+7j)
Conjugate: (1-7j)
```

3 Complex expressions equations

AIM: Learning to work with and evaluate complex expressions and equations.

3.1 Euler's formula to evaluate e^{x+iy}

Euler's formula states that $e^{iy} = cos(y) + isin(y)$ Hence, by Euler's formula$e^{x+iy} = e^x e^{iy} = e^x(cos(y) + isin(y))$.

[75]:
```
z = complex(input(">> "))
```

```
>> 4-2j
```

[76]:
```
def eulersFormula(z):
    e_power_x = cmath.e**z.real # Can also use cmath.exp(z.real)
    e_power_iy = cmath.cos(z.imag) + complex(0, 1)*cmath.sin(z.imag)
    return e_power_x * e_power_iy

print("e^z (using inbuilt CMath function):", cmath.e**z) # Can also use␣
↪cmath.exp(z)
print("e^z (Euler's formula):", eulersFormula(z))
```

```
e^z (using inbuilt CMath function): (-22.72084741761923-49.64595733458056j)
e^z (Euler's formula): (-22.72084741761923-49.64595733458056j)
```

3.2 Finding point furthest from origin

The magnitude of a complex number gives its distance from the origin. Hence, to see which point is furthest from the origin, we must see which point has the greatest magnitude.

QUESTION: Out of the three points, $z_1 = 2.5 + 1.9j$, $z_2 = 1.5 - 2.9j$ and $z_3 = -2 + 2.2j$, find the point which is farthest away from the origin

[24]:
```
complexNumbers = [complex(2.5, 1.9), # z1
                  complex(1.5, 2.9), # z2
                  complex(-2, 2.2)]  # z3
```

[42]:
```python
# METHOD 1
# Finding complex number with largest magnitude
zmax = complexNumbers[0]
for z in complexNumbers:
    if abs(z) > abs(zmax):
        zmax = z
print("Complex number with most distance from origin:", zmax)
```

```
Complex number with most distance from origin: (1.5+2.9j)
```

[47]:
```python
# METHOD 2
# List of magnitudes
magnitudes = list(map(abs, complexNumbers))
zmax = complexNumbers[magnitudes.index(max(magnitudes))]
print("Complex number with most distance from origin:", zmax)
```

```
Complex number with most distance from origin: (1.5+2.9j)
```

3.3 Solving simple complex equations

If $a + ib = \frac{3-i}{2+3i} + \frac{2-2i}{1-5i}$, find a and b .

[49]:
```python
# First, we must compute the complex expression.
# Here, a is the real part and b is the imaginary part of this complex␣
↪expression.
complexExpression = complex(3, 1) / complex(2, 3) + complex(2, -2) /␣
↪complex(1, -5)
print("a =", complexExpression.real)
print("b =", complexExpression.imag)
```

```
a = 1.153846153846154
b = -0.23076923076923084
```

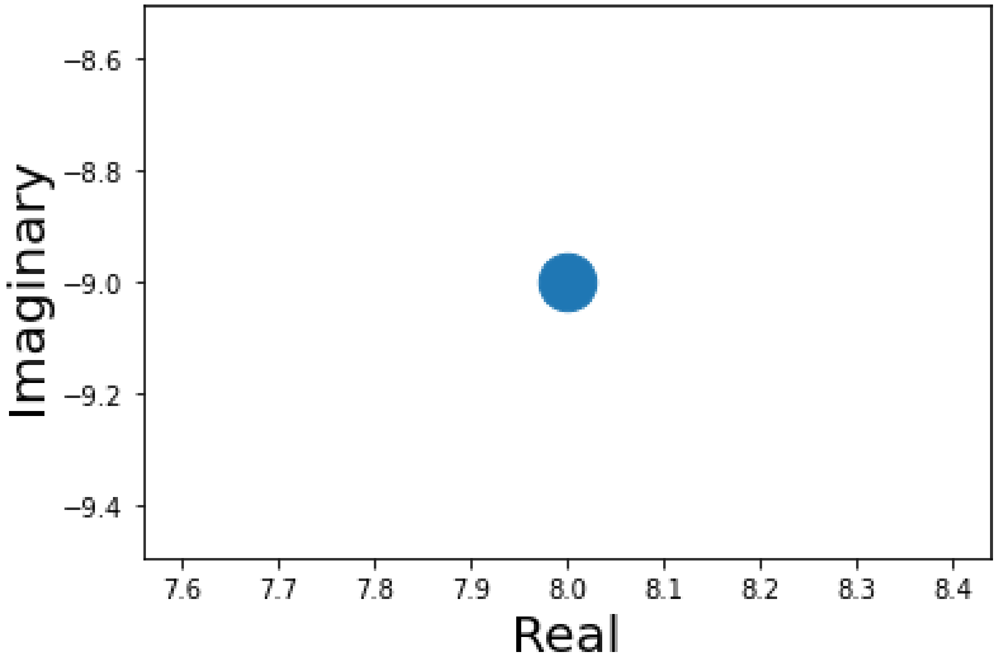
4 Plotting complex numbers roots of unity

AIM: Learning to plot complex numbers and functions, as well as finding and plotting the nth roots of unity.

[1]:
```python
import cmath
import matplotlib.pyplot as plt
```

4.1 Plotting single complex number

[2]:
```
z = complex("8-9j")
plt.scatter(z.real, z.imag, marker = 'o', s = 500)
# s gives marker size.
# s may be the same size as x or y (to size points according to x or y␣
↪values).
plt.xlabel("Real", fontsize = 20)       # Extra
plt.ylabel("Imaginary", fontsize = 20)  # Extra
plt.show()                              # Extra
```



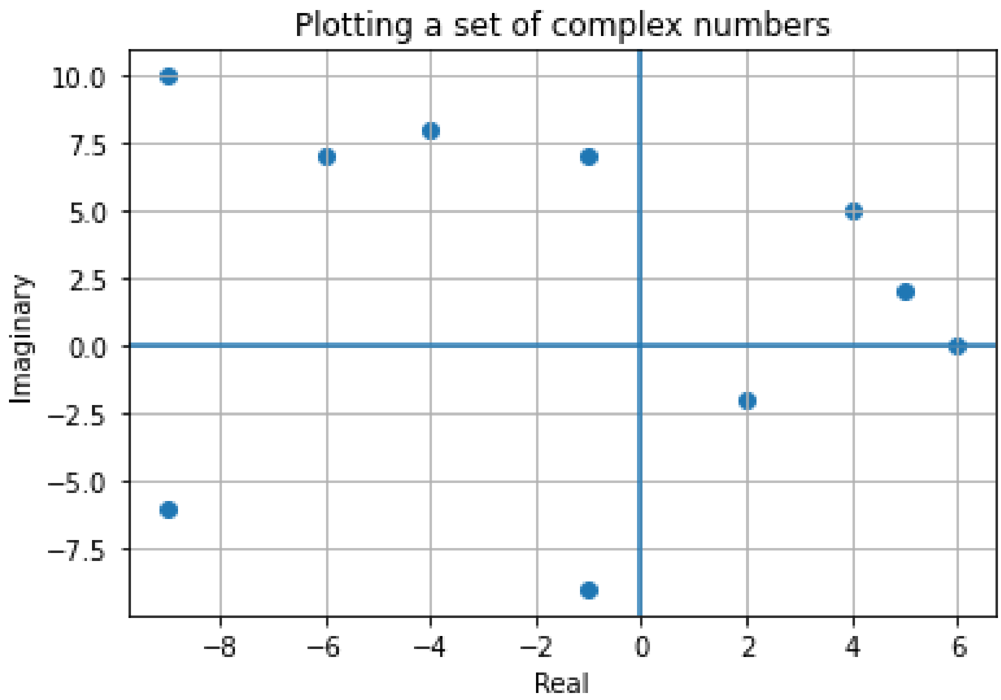
4.2 Plotting a set of complex numbers

[3]:
```
# Generating a random set of ten complex numbers
from random import randint
Z = []
for i in range(0, 10):
    Z.append(complex(randint(-10, 10), randint(-10, 10)))

# Printing the list of complex numbers
```

```
print(Z)
```

```
[(-9+10j), (6+0j), (-4+8j), (-6+7j), (-1+7j), (-1-9j), (-9-6j), (4+5j),␣
↪(5+2j),
(2-2j)]
```

[4]:
```
# Functions to return the real and imaginary parts of a complex number
# (Functions are created to use in the 'map' function)
def real(z): return z.real
def imag(z): return z.imag

# Creating x and y values
re = list(map(real, Z))
im = list(map(imag, Z))

# Plotting the complex numbers
plt.scatter(re, im, marker = 'o')
plt.title("Plotting a set of complex numbers")
plt.xlabel("Real")
plt.ylabel("Imaginary")

# To show grid and axes
plt.plot(0, 0)
plt.grid()
plt.axvline()
plt.axhline()

# Displaying the plot alone
plt.show()
```



4.3 Plotting cube roots of unity

Cube roots of unity are the solutions of the equation $a^3 = 1$. We will find these solutions using SymPy, then use these solutions to plot points on the complex plane.

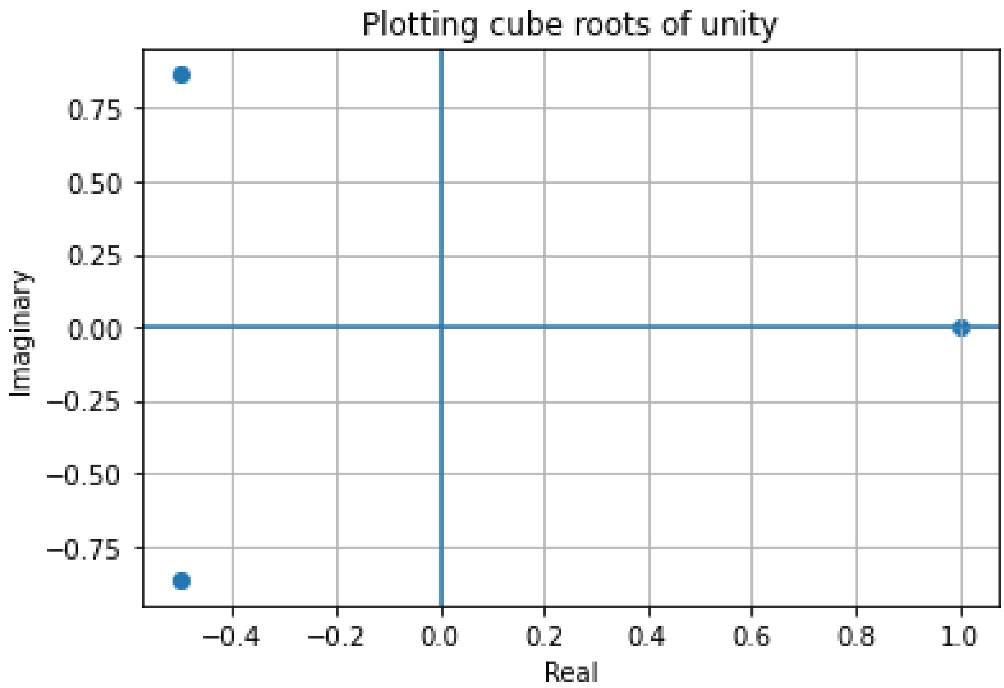
[7]:
```
# FINDING CUBE ROOTS OF UNITY
#-------------------------
# Obtaining cube roots of unity (denoted here as cbou)
import sympy as sp
a = sp.Symbol('a')
equation = sp.Eq(a**3, 1)
cbou = sp.solve(equation, a)

# Printing the cube roots of unity obtained
print("\nObtained cube roots of unity:")
for r in cbou: print(r)

# Converting the list of cube roots of unity to complex numbers
cbou = list(map(complex, cbou))
print("\nCube roots of unity as complex numbers:")
```

```
for r in cbou: print(r)
```

```
Obtained cube roots of unity:
1
-1/2 - sqrt(3)*I/2
-1/2 + sqrt(3)*I/2

Cube roots of unity as complex numbers:
(1+0j)
(-0.5-0.8660254037844386j)
(-0.5+0.8660254037844386j)
```

[6]:

```
# PLOTTING THE ROOTS
#------------------------
# Finding the list of real and imaginary parts separately
def real(z): return z.real
def imag(z): return z.imag
re, im = list(map(real, cbou)), list(map(imag, cbou))

# Plotting the cube roots of unity
plt.scatter(re, im)

# Adding additional plot elements
plt.title("Plotting cube roots of unity")
plt.xlabel("Real")
plt.ylabel("Imaginary")

# To show grid and axes
plt.grid()
plt.axvline()
plt.axhline()

# Displaying the plot alone
plt.show()
```



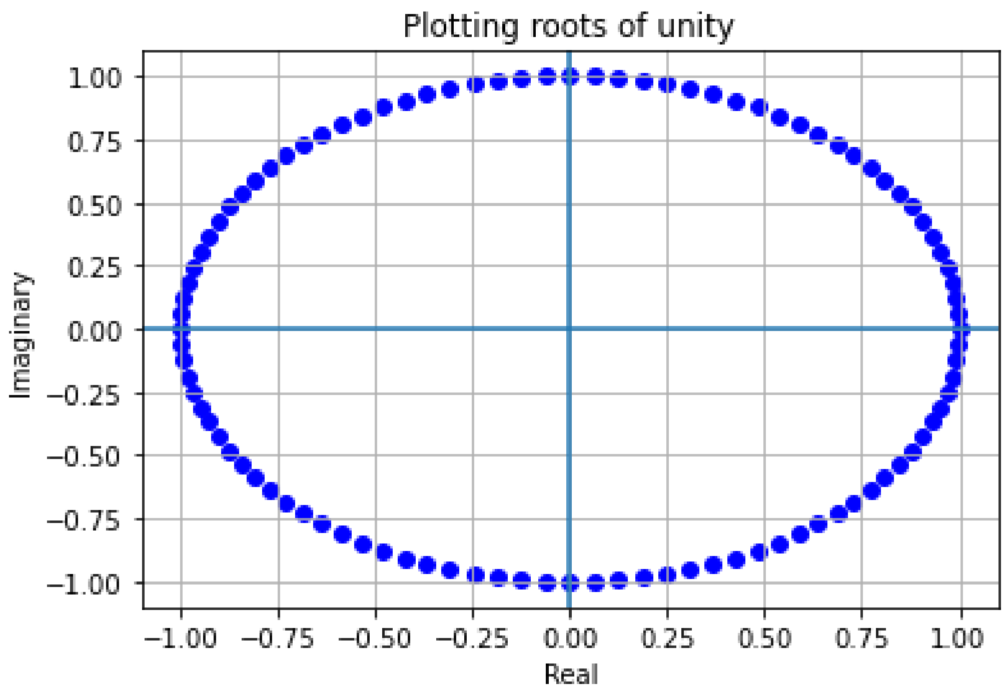
4.4 Plotting nth roots of unity

[121]:
```
# FINDING N ROOTS OF UNITY
#------------------------
# Inputting n
n = abs(int(input("n = ")))

# Obtaining cube roots of unity (denoted here as cbou)
import sympy as sp
a = sp.Symbol('a')
equation = sp.Eq(a**n, 1)
cbou = sp.solve(equation, a)

# Converting the list of cube roots of unity to complex numbers
cbou = list(map(complex, cbou))
```

```
n = 100
```

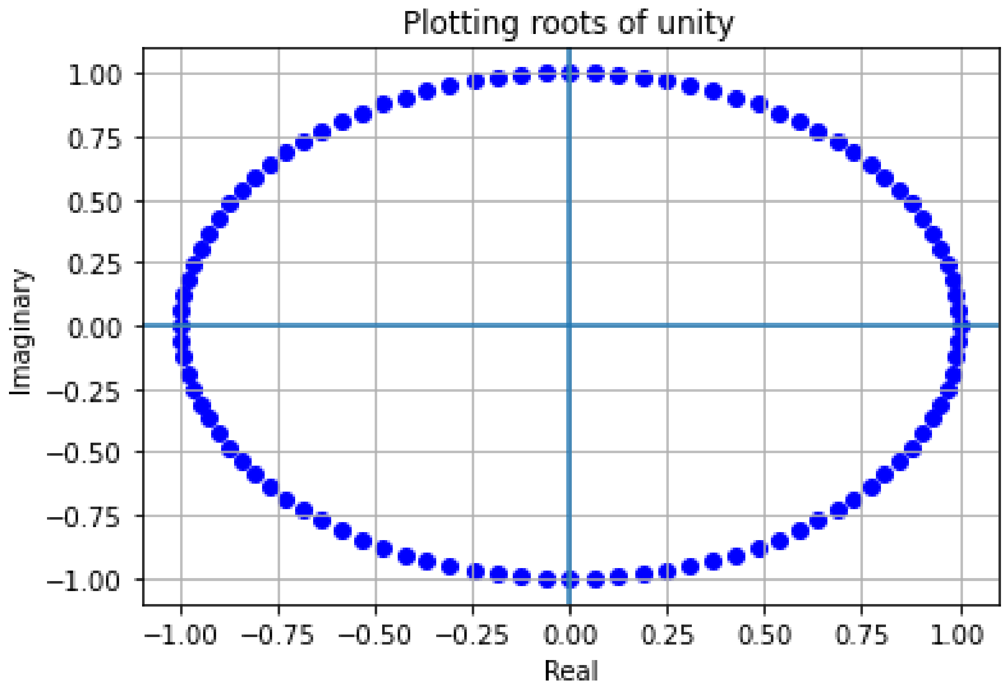
[123]:
```
# PLOTTING THE ROOTS
#------------------------
```

```
# Finding the list of real and imaginary parts separately
def real(z): return z.real
def imag(z): return z.imag
re, im = list(map(real, cbou)), list(map(imag, cbou))

# Plotting the cube roots of unity
plt.scatter(re, im, color = "blue")
plt.title("Plotting roots of unity")
plt.xlabel("Real")
plt.ylabel("Imaginary")

# To show grid and axes
plt.grid()
plt.axvline()
plt.axhline()

# Displaying the plot alone
plt.show()
```



NOTE ON PLOTTING METHOD: You could use the above method, where you generate the corresponding lists of real and imaginary parts for each root of unity, then plot the lists, Or, you could plot each complex number separately, as shown below. The latter method is notably more time consuming, which becomes apparent for larger values of n.

[119]:
```
# Plotting the cube roots of unity
for z in cbou:
    plt.scatter(z.real, z.imag, color = "blue")

# Adding additional plot elements
plt.title("Plotting roots of unity")
plt.xlabel("Real")
plt.ylabel("Imaginary")

# To show grid and axes
plt.grid()
plt.axvline()
plt.axhline()

# Displaying the plot alone
plt.show()
```



5 Limits of complex sequences

AIM: Find limit of a complex function / sequence

[1]:
```
import cmath as c
import math as m
import sympy as sp
import matplotlib.pyplot as plt
```

Syntax of the limit function in SymPy is **limit(exp, var, c)** where exp is the expression in terms of symbols var is the symbol denoting the variable approaching a constant c is the constant being approached

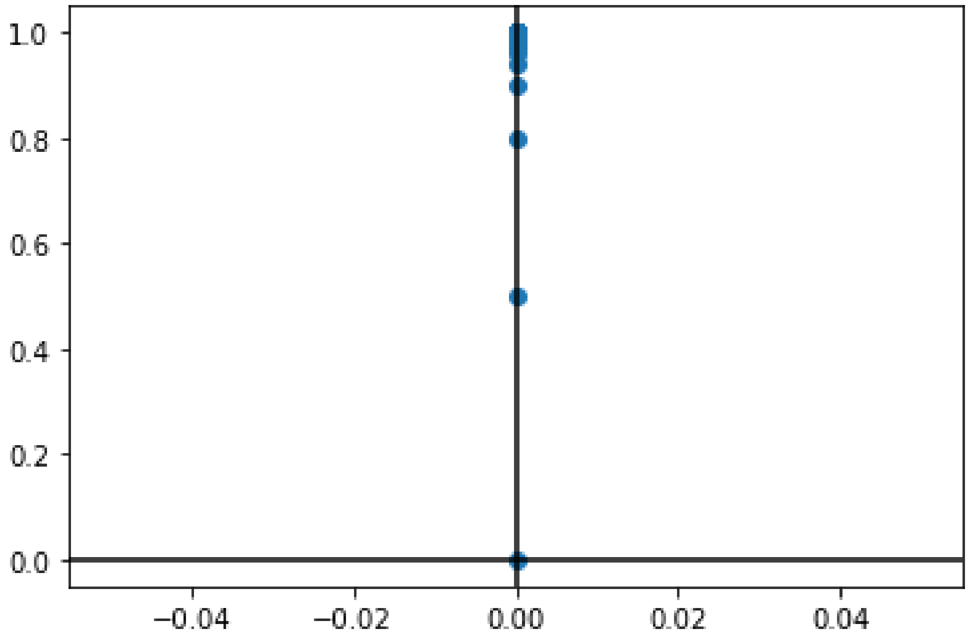
5.1 Limit of $i\frac{n^2}{n^2+1}$ as n approaches infinity

[2]:
```
n = sp.Symbol('n')
im_exp = n**2 / (n**2 + 1)

# Since we have only imaginary part, real part is 0
complex(0, sp.limit(im_exp, n, float('inf')))
```

[2]:
```
1j
```

5.1.1 Plotting the above function

[3]:
```
lower, upper = 0, 100
re, im = [], []
for x in range(lower, upper):
    re.append(0)
    im.append(x**2 / (x**2 + 1))

# Plotting the graph
plt.scatter(re, im)
plt.axvline(color = "black")
plt.axhline(color = "black")
plt.show()
```



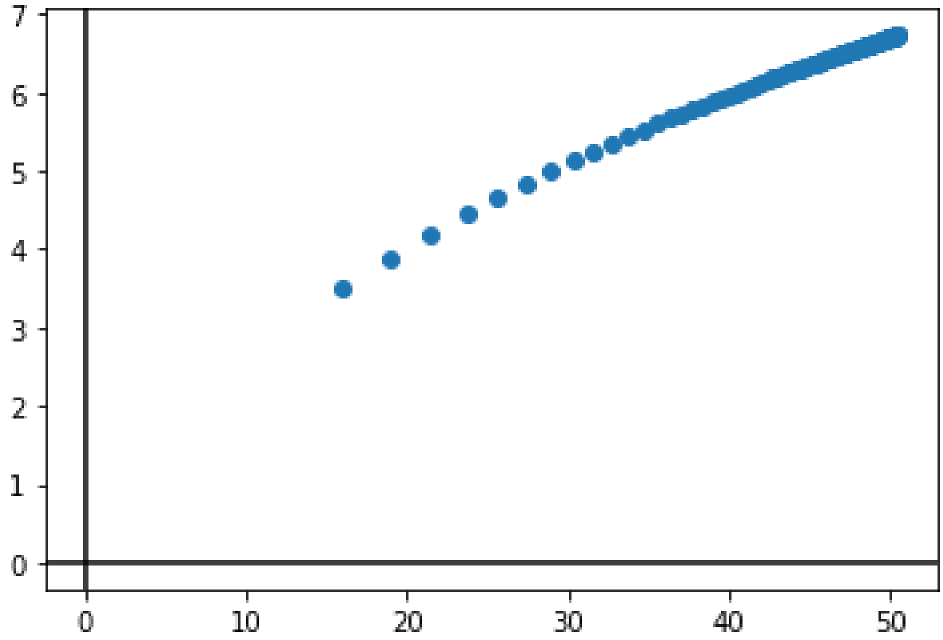
5.2 Limit of $(1 + \frac{4}{n})^n + i\frac{7n}{n+4}$ as n approaches infinity

[4]:
```
n = sp.Symbol('n')
re_exp = (1 + 4/n)**n
im_exp = 7*n/(n + 4)
inf = float('inf')

# Since we have only imaginary part, real part is 0
complex(sp.limit(re_exp, n, inf), sp.limit(im_exp, n, inf))
```

[4]: (54.598150033144236+7j)

[5]:
```
lower, upper = 4, 100
re, im = [], []
for x in range(lower, upper):
    re.append((1 + 4/x)**x)
    im.append(7*x/(x + 4))

# Plotting the graph
plt.scatter(re, im)
plt.axvline(color = "black")
```

```
plt.axhline(color = "black")
plt.show()
```



5.2.1 Plotting the above using polar plot

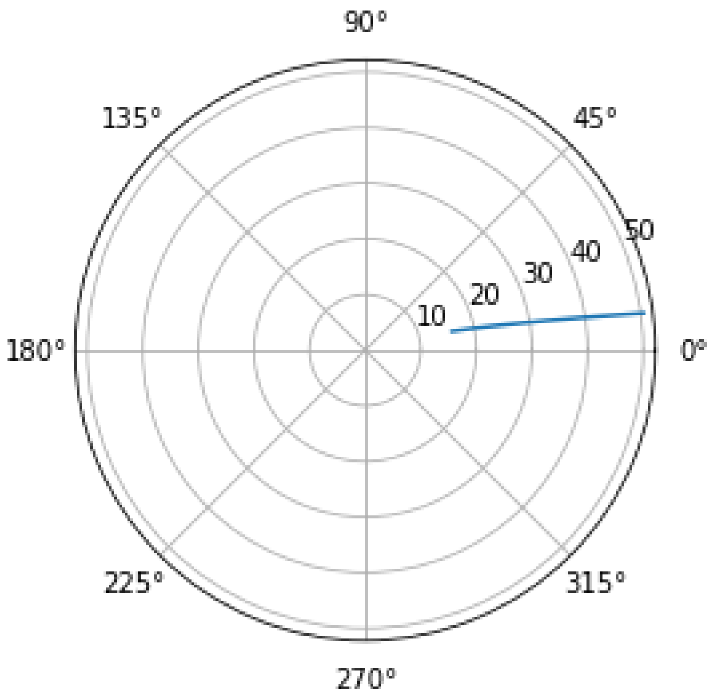
[6]:
```
# Obtaining polar coordinate expressions
# Modulus
r = sp.sqrt(re_exp**2, im_exp**2)
# Argument
theta = sp.atan(im_exp/re_exp)
```

[7]:
```
# Finding limit in polar form
# z = re^(i*theta)
inf = float('inf')
limr = float(sp.limit(r, n, inf))
limtheta = float(sp.limit(theta, n, inf))
print("lim(r): ", limr)
print("lim(theta) (in radians):", limtheta)
print("lim(theta) (in degrees):", 180*limtheta/m.pi)
```

```
lim(r):  54.598150033144236
```

```
lim(theta) (in radians): 0.1275138319828531
lim(theta) (in degrees): 7.306004402157776
```

[8]:
```python
# Obtaining polar coordinates for different values of n
lower, upper = 4, 100
rs, thetas = [], []
for x in range(lower, upper):
    rs.append(eval(str(r.subs({n: x}))))
    thetas.append(eval(str(theta.subs({n: x})), {'atan': m.atan}))
"""
Here, I have associated the word 'atan' with the atan function from the␣
 ↪Math module.
Such associations can be done for functions and variables using the␣
 ↪second argument.
The second argument is a dictionary associating various strings with␣
 ↪their values.
This is done since atan is not available in default Python.

NOTE: eval only accepts strings
"""

# Plotting the graph
# Syntax of polar plot: polar(theta, r, etc.)
plt.polar(thetas, rs)
# NOTE: thetas i.e. arguments must be given in radians
plt.show()
```



5.3 Limit properties verification

5.3.1 Limit of sum of sequences = Sum of limits of sequences

Confirm for $\frac{1}{n} + i\frac{n-1}{n}$ and $i\frac{n^2}{n^2+1}$

[23]:
```
# Sequence 1:
n = sp.Symbol('n')
re_exp_1 = 1/n
im_exp_1 = (n-1)/n

# Sequence 2:
n = sp.Symbol('n')
re_exp_2 = 0
im_exp_2 = (n**2)/(n**2 + 1)

# Sum of sequences:
re_exp_sum = re_exp_1 + re_exp_2
im_exp_sum = im_exp_1 + im_exp_2
```

Limit of sum of sequences…

[24]:
```
inf = float('inf')
re_limit = sp.limit(re_exp_sum, n, inf)
im_limit = sp.limit(im_exp_sum, n, inf)
print(complex(re_limit, im_limit))
```

```
2j
```

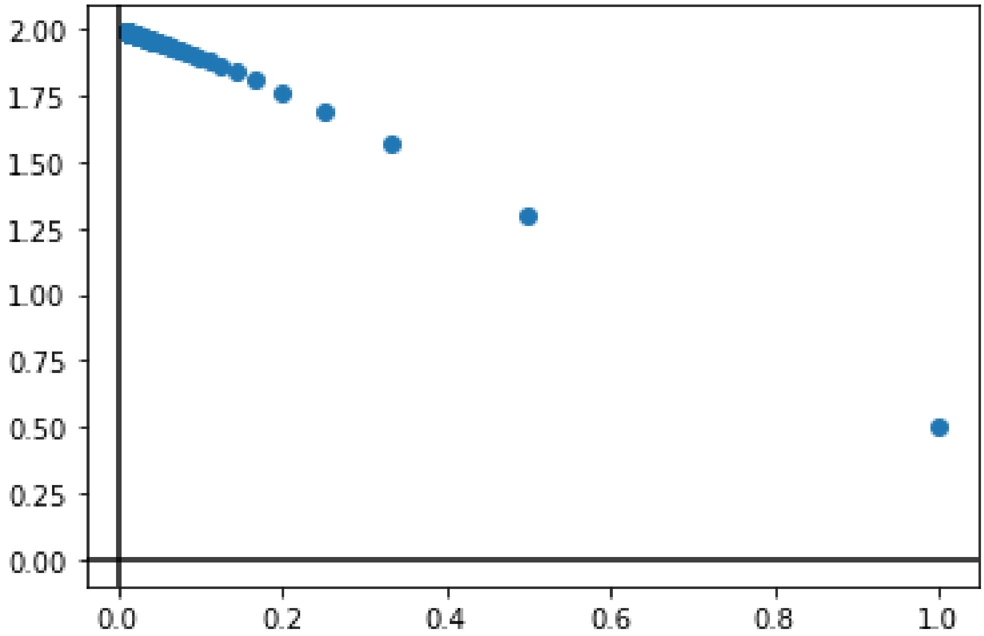
Sum of limits of sequences

[22]:
```
# Limit of sequence 1:
lim_1 = complex(sp.limit(re_exp_1, n, inf), sp.limit(im_exp_1, n, inf))
print("Limit of sequence 1: ", lim_1)

# Limit of sequence 2:
lim_2 = complex(sp.limit(re_exp_2, n, inf), sp.limit(im_exp_2, n, inf))
print("Limit of sequence 2: ", lim_2)

# Sequence 1:
print("Sum of limits of sequences: ", lim_1 + lim_2)
```

```
Limit of sequence 1:  1j
Limit of sequence 2:  1j
Sum of limits of sequences:  2j
```

Plotting sum of sequences

[37]:
```
lower, upper = 1, 100
re, im = [], []
for x in range(lower, upper):
    re.append(eval(str(re_exp_sum.subs({n: x}))))
    im.append(eval(str(im_exp_sum.subs({n: x}))))

# Plotting the graph
plt.scatter(re, im)
plt.axvline(color = "black")
plt.axhline(color = "black")
plt.show()
```



5.3.2 k • limit of of sequence = limit of k • sequence (k is a constant)

Confirm for $\frac{1}{n} + i\frac{n-1}{n}$

[48]:
```
# Real and imaginary expressions
n = sp.Symbol('n')
re_exp = 1/n
im_exp = (n-1)/n

# Inputting complex constant
k = complex(input("k = "))

# k * limit of sequence
lim_1 = k * complex(sp.limit(re_exp, n, inf), sp.limit(im_exp, n, inf))
print("k • limit of sequence:", lim_1)

# Limit of k • sequence
lim_2 = complex(sp.limit(k*re_exp, n, inf), sp.limit(k*im_exp, n, inf))
print("Limit of k • sequence:", lim_2)
```

```
k = 3
k • limit of sequence: 3j
```

```
Limit of k ● sequence: 3j
```

6 Cauchy-Riemann equations

AIM: Verifying analytic functions using Caucy-Riemann equations

A complex function $f(z) = u(x, y) + iv(x, y)$ (where $z = x + iy$) is analytic if and only if the following equations are satisfied: $\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y}$ and $\frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x}$. These are the Cauchy-Riemann equations.

6.1 Necessary libraries

[1]:
```
import sympy as sp
```

6.2 Check if $f(z) = x^2 - y^2 + 2xyi$ is analytic

[2]:
```
# Defining variable symbols
x = sp.Symbol('x')
y = sp.Symbol('y')

# Defining u(x, y) (real part) and v(x, y) (imaginary part)
u = x**2 - y**2
v = 2*x*y
```

[3]:
```
# Defining a function to test whether the Cauchy-Riemann equations are␣
↪satisfied
# (If satisfied, it means the complex function is analytic)
def isAnalytic(u, v, x, y):
    # Obtaining partial derivatives
    du_dx = sp.diff(u, x)
    du_dy = sp.diff(u, y)

    dv_dx = sp.diff(v, x)
    dv_dy = sp.diff(v, y)
    #------------------------
    # Displaying the obtained partial derivatives
    print("Partial derivative of u w.r.t. x:")
    print(du_dx)
    print("Partial derivative of u w.r.t. y:")
    print(du_dy)
    print("------------------------")
```

```
    print("Partial derivative of v w.r.t. x:")
    print(dv_dx)
    print("Partial derivative of v w.r.t. x:")
    print(dv_dy)
    print("------------------------")
    #------------------------
    # Confirming Cauchy-Riemann equations
    print("Cauchy-Riemann equations are satisfied?", end = " ")
    print(du_dx == dv_dy and du_dy == -dv_dx)
```

[4]:
```
isAnalytic(u, v, x, y)
```

```
Partial derivative of u w.r.t. x:
2*x
Partial derivative of u w.r.t. y:
-2*y
------------------------
Partial derivative of v w.r.t. x:
2*y
Partial derivative of v w.r.t. x:
2*x
------------------------
Cauchy-Riemann equations are satisfied? True
```

Since the Cauchy-Riemann equations are satisfied, we can conclude that $f(z)$ is analytic.

6.3 Check if $f(z) = 2xy + 2xi$ is analytic

[5]:
```
# Defining variable symbols
x = sp.Symbol('x')
y = sp.Symbol('y')

# Defining u(x, y) (real part) and v(x, y) (imaginary part)
u = 2*x*y
v = 2*x

# Checking if Cauchy-Riemann equations are satisfied
isAnalytic(u, v, x, y)
```

```
Partial derivative of u w.r.t. x:
2*y
Partial derivative of u w.r.t. y:
2*x
------------------------
```

```
Partial derivative of v w.r.t. x:
2
Partial derivative of v w.r.t. x:
0
------------------------
Cauchy-Riemann equations are satisfied? False
```

Since the Cauchy-Riemann equations are not satisfied, we can conclude that $f(z)$ is not analytic.

6.4 Check if $f(z) = \frac{x-iy}{x^2+y^2}$ is analytic

[6]:
```python
# Defining variable symbols
x = sp.Symbol('x')
y = sp.Symbol('y')

# Defining u(x, y) (real part) and v(x, y) (imaginary part)
u = x/(x**2 + y**2)
v = -y/(x**2 + y**2)

# Checking if Cauchy-Riemann equations are satisfied
isAnalytic(u, v, x, y)
```

```
Partial derivative of u w.r.t. x:
-2*x**2/(x**2 + y**2)**2 + 1/(x**2 + y**2)
Partial derivative of u w.r.t. y:
-2*x*y/(x**2 + y**2)**2
------------------------
Partial derivative of v w.r.t. x:
2*x*y/(x**2 + y**2)**2
Partial derivative of v w.r.t. x:
2*y**2/(x**2 + y**2)**2 - 1/(x**2 + y**2)
------------------------
Cauchy-Riemann equations are satisfied? False
```

Since the Cauchy-Riemann equations are not satisfied, we can conclude that $f(z)$ is not analytic.

7 Harmonic functions & conjugates

A function $u(x,y)$ is said to be harmonic if it satisfies the Laplace equation i.e. $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 v}{\partial x^2} = 0$

[1]:
```python
import sympy as sp
def isHarmonic(function, variables):
    summation = 0
    print("------------\nFunction:")
    print(f)
    for var in variables:
        # Calculating the partial derivative of order 2
        partialDerivative = function.diff(var, 2)
        print("Partial derivative w.r.t.", var, end = ":\n")
        print(partialDerivative)

        # Adding to the cumulative
        summation = summation + partialDerivative
    # Forcing expression evaluation through 'sp.simplify'
    summation = sp.simplify(summation)

    # Returning a Boolean
    return summation == 0
```

[2]:
```python
x, y, = sp.Symbol('x'), sp.Symbol('y')
functions = [
    2*x*y,
    sp.log(sp.sqrt(x**2 + y**2)),
    sp.atan(y/x)]
for f in functions:
    print("Harmonic?",  isHarmonic(f, (x, y)))
```

```
------------
Function:
2*x*y
Partial derivative w.r.t. x:
0
Partial derivative w.r.t. y:
0
Harmonic? True
------------
Function:
log(sqrt(x**2 + y**2))
Partial derivative w.r.t. x:
(-2*x**2/(x**2 + y**2) + 1)/(x**2 + y**2)
Partial derivative w.r.t. y:
(-2*y**2/(x**2 + y**2) + 1)/(x**2 + y**2)
Harmonic? True
```

```
------------
Function:
atan(y/x)
Partial derivative w.r.t. x:
2*y*(1 - y**2/(x**2*(1 + y**2/x**2)))/(x**3*(1 + y**2/x**2))
Partial derivative w.r.t. y:
-2*y/(x**3*(1 + y**2/x**2)**2)
Harmonic? True
```

7.1 Harmonic conjugates

If $u(x, y)$ is a harmonic function, its harmonic conjugate is another harmonic function $v(x, y)$ such that they satisfy the Cauchy-Riemann equatioins in the following manner: $\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y}$, $\frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x}$

[14]:
```python
# Redefining the 'isHarmonic' function without the print statements
def isHarmonic(function, variables):
    summation = 0
    for var in variables:
        # Calculating the partial derivative of order 2
        partialDerivative = function.diff(var, 2)

        # Adding to the cumulative
        summation = summation + partialDerivative
    # Forcing expression evaluation through 'sp.simplify'
    summation = sp.simplify(summation)

    # Returning a Boolean
    return summation == 0
#========================
# Finding harmonic conjugate
def harmonicConjugate(function, variables):
    if not isHarmonic(function, variables):
        return "Given function is not harmonic!"

    # Finding partial derivatives of 'function'
    du_dx = function.diff(x)
    du_dy = function.diff(y)

    # Applying Cauchy-Riemann equations for an unknown function v
    dv_dx = -du_dy
    dv_dy = du_dx
    """
```

```
    The above assignment operations are merely done for conceptual␣
↪clarity.
    They are not practically necessary, and these steps can be reduced.
    """

    # Obtaining and simplifying v
    v = (dv_dx.integrate(x) + dv_dy.integrate(y)) / 2
    """
    The integrals are divided by 2 based on what led to the correct␣
↪results.
    The exact reasoning is yet unknown to me.
    """
    v.simplify()
    return v
```

[15]:
```
x, y, z = sp.Symbol('x'), sp.Symbol('y'), sp.Symbol('z')
u = sp.log(x**2+y**2)/2
u
```

[15]:
$$\frac{\log\left(x^2+y^2\right)}{2}$$

[17]:
```
harmonicConjugate(u, (x, y))
```

[17]:
$$\frac{i\log\left(x-iy\right)}{4}-\frac{i\log\left(x+iy\right)}{4}-\frac{i\log\left(-ix+y\right)}{4}+\frac{i\log\left(ix+y\right)}{4}$$

8 Milne-Thompson method

This is a method to find the complex function (in terms of z) whose real or imaginary part is given. If real part $u(x,y)$ is given, we use the equation $f'(z)=\frac{\partial u}{\partial x}-i\frac{\partial u}{\partial y}$. If imaginary part $v(x,y)$, we use the equation $f'(z)=\frac{\partial v}{\partial y}+i\frac{\partial v}{\partial x}$

[1]:
```
import sympy as sp
def milne_thompson(function, variables, isReal):
    if isReal: df_dz = sp.diff(function, x) - sp.I*sp.diff(function, y)
    else: df_dz = sp.diff(function, y) + sp.I*sp.diff(function, x)
    # Substituting x = z and y = 0
    df_dz = df_dz.subs({x:z, y:0})
    # Obtaining f(z)
    f = df_dz.integrate(z)
    return f
```

[2]:
```python
x, y, z = sp.Symbol('x'), sp.Symbol('y'), sp.Symbol('z')
u = sp.log(sp.sqrt(x**2 + y**2))
milne_thompson(u, (x, y), True)
```

[2]: $\log(z)$

8.1 With user input

[3]:
```python
def milne_thompson_with_input():
    function = input("Enter function:\n")
    isReal = input("For f(z) = u + iv, is the above function u or v? ")
    if isReal == "u": isReal = True
    elif isReal == "v": isReal = False
    else:
        print("Invalid inputs!")
        return

    # Recognising variables...
    variables = []
    for c in function:
        if c.isalpha() and c not in variables: variables.append(sp.
↪Symbol(c))

    # It seems that you can apply SymPy functions on strings as well.
    # (as long as the required symbols are defined)

    # Applying Milne-Thompson method...
    print("The above function in terms of z:")
    print(milne_thompson(function, variables, isReal))
```

[4]:
```python
milne_thompson_with_input()
```

```
Enter function:
x**2-y**2
For f(z) = u + iv, is the above function u or v? u
The above function in terms of z:
z**2
```

9 Power series

9.1 Taylor series expansion

The nth degree Taylor's series expansion for $f(z)$ is given by $\sum_0^n \frac{f^k(z_0)(z-z_0)^k}{k!}$, which is an approximation of $f(z)$ around the point $z = z_0$.

[1]:
```
from sympy import *
# HELPER FUNCTION
def expression(expansion):
    expansion = '+'.join(list(map(str, expansion)))
    expansion = sympify(expansion)
    return expansion

# MAIN FUNCTION
def taylor(f, z, c, n, returnType):
    z = Symbol(z)
    expansion, prevDiff, factorial = [], f, 1
    # prevDiff contains the previous derivative of f.
    # So, if we are at the kth iteration, prevDiff contains the (k-1)th␣
↪derivate of f.
    # This is to reduce computational cost.

    # factorial contains the factorial of the previous k value.
    # This is also to reduce computational cost.
    for i in range(0, n+1):
        tmp = prevDiff.subs({z:c})
        if float(tmp) != 0: expansion.append((tmp*(z-c)**i)/factorial)

        prevDiff = prevDiff.diff()
        factorial = factorial*(i+1)

    # Processing expansion based on return type option
    return {"expression": expression,
            "list": list,
            "tuple": tuple}[returnType](expansion)
```

[2]:
```
taylor(exp('z'), 'z', 0, 5, 'expression')
```

[2]:
$$\frac{z^5}{120} + \frac{z^4}{24} + \frac{z^3}{6} + \frac{z^2}{2} + z + 1$$

[3]:
```
taylor(sin('z'), 'z', pi, 5, 'expression')
```

[3]:

$$-z - \frac{(z-\pi)^5}{120} + \frac{(z-\pi)^3}{6} + \pi$$

9.2 Radius of convergence

An infinite series of the form $S = \sum_0^n a_k(z-z_0)^k$ is called a power series centred at z_0 (centre of the series). The circle $|z - z_0| = r \in R_+$ such that S converges for every set of points within this crcle is called the circle of convergence of the series, and its radius is called the radius of convergence of the series.

For the series $S = \sum_0^\infty a_k(z-z_0)^k$, the radius of convergence R can be evaluated as follows…

- If $\lim_{n\to\infty} |\frac{a_{n+1}}{a_n}| = L \neq 0$, then $R = \frac{1}{L}$
- If $\lim_{n\to\infty} |\frac{a_{n+1}}{a_n}| = 0 \neq 0$, then $R = \infty$
- If $\lim_{n\to\infty} |\frac{a_{n+1}}{a_n}| = \infty \neq 0$, then $R = 0$

Alternatively, if $\lim_{n\to\infty} \sqrt[n]{a_n} = L$, then $R = \frac{1}{L}$

[28]:
```
# Function to find radius of convergence
def roc(a_k, method):
    """
    a_k is the expression of the kth coefficient of the series.
    Note that it must be defined considering
    k as the iterating variable.

    NOTE: a_k must b given as an expression string.

    method is a number that specifies whether you want the
    program to evaluate the radius of convergence using
    the 1st formula or the 2nd formula
    """
    a_k = sympify(a_k)
    k = Symbol('k')

    # Limit expression: a_k / a_(k-1)
    if method == 1: a = abs(a_k/a_k.subs({k:k-1}))

    # Limit expression: (a_k)^(1/k)
    elif method == 2: a = (a_k)**(1/k)

    # Trying to evaluate limit
    try: L = limit(a, k, float("inf"))
    except: return "Could not calculate!"
```

```
    return 1/L
```

Applying the above function…

[32]:
```
roc("1/(1-2*I)^(k+1)", 1)
```

[32]: $\sqrt{5}$

[33]:
```
roc("1/(1-2*I)^(k+1)", 2)
```

[33]: 'Could not calculate!'

[36]:
```
roc("((6*k+1)/(2*k+5))^k", 1)
```

[36]: $\frac{1}{3}$

[37]:
```
roc("((6*k+1)/(2*k+5))^k", 2)
```

[37]: $\frac{1}{3}$

As we can see, both methods can be used if many cases, though in some cases, using one may lead to a computational error. Hence, in the actual implementation of this radius of convergence finder, we can simply try one method, and if it does not work, try the other…

10 Elementary transformations

[1]:
```
from random import randint
from matplotlib.pyplot import scatter, plot, show, axhline, axvline, axis
```

[2]:
```
def inputComplex(prompt):
    try: return complex(input(prompt))
    except:
        print("Not a complex number! Defaulting to 0.")
        return complex(0, 0)

def inputReal(prompt):
    try: return float(input(prompt))
    except:
        print("Not a real number! Defaulting to 0.")
        return 0

def graph(z, w):
```

```
    scatter([z.real, w.real], [z.imag, w.imag])
    axhline()
    axvline()
    axis('square')
    show()

def graphWithConnection(z, w):
    scatter([z.real, w.real], [z.imag, w.imag])
    plot([z.real, w.real], [z.imag, w.imag])
    axhline()
    axvline()
    axis('square')
    show()

def graphWithOriginVectors(z, w):
    scatter([z.real, w.real], [z.imag, w.imag])
    plot([0, z.real], [0, z.imag], color = "red")
    plot([0, w.real], [0, w.imag], color = "blue")
    axhline()
    axvline()
    axis('square')
    show()
```

10.1 Translation

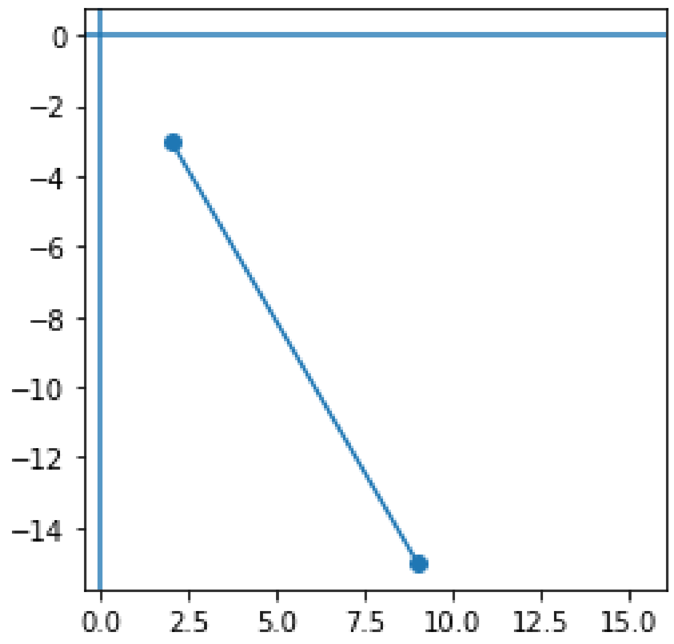
$w = z + k,\ k \in R$

[3]:
```
def translate(z, k): return z + k
```

[4]:
```
z1 = inputComplex("Preimage = ")
k = inputComplex("Translate by: ")
w1 = translate(z1, k)
print("Image =", w1)
```

```
Preimage = 2-3j
Translate by: 7-12j
Image = (9-15j)
```

[6]:
```
graphWithConnection(z1, w1)
```



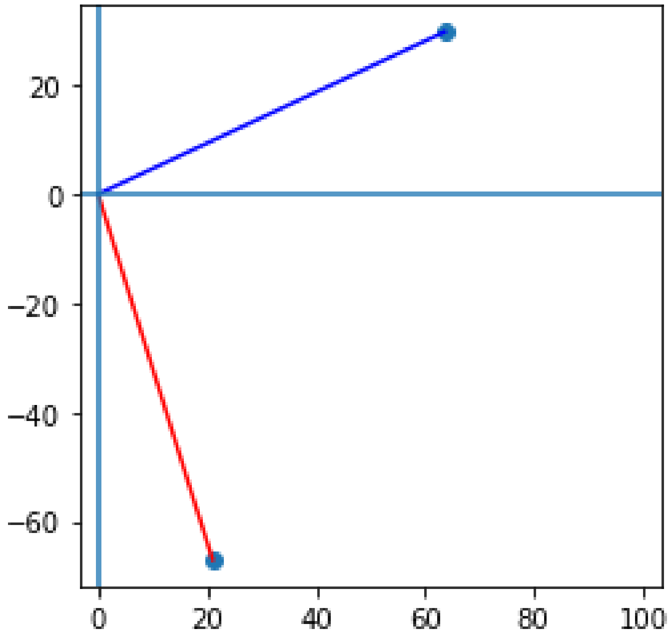
10.2 Rotation

[8]:
```
from numpy import e
def rotate(z, theta): return z*(e**(complex(0, 1)*theta))
```

[9]:
```
z2 = inputComplex("Preimage = ")
k = inputReal("Rotate by: ")
w2 = rotate(z2, k)
print("Image =", w2)
```

```
Preimage = 21-67j
Rotate by: 1.705
Image = (63.58772412718167+29.775851630565278j)
```

[11]:
```
graphWithOriginVectors(z2, w2)
```



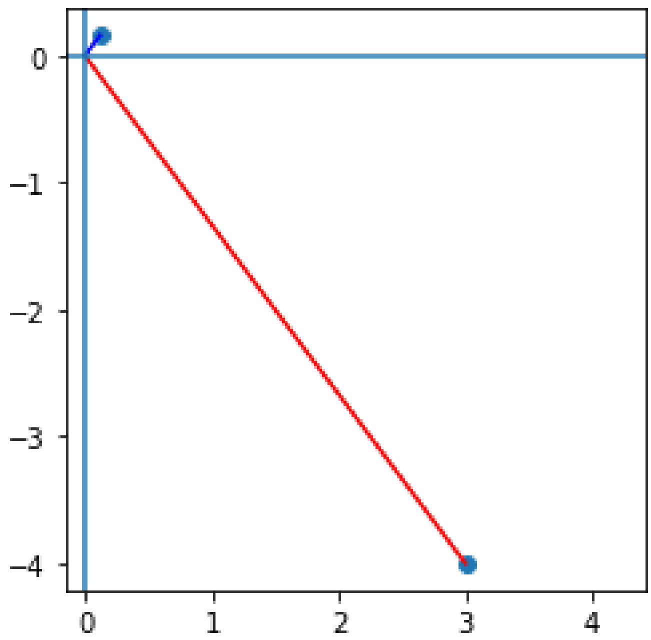
10.3 Inversion

[12]:
```
def invert(z): return 1/z
```

[15]:
```
z3 = inputComplex("Preimage = ")
w3 = invert(z3)
print("Image =", w3)
```

```
Preimage = 3-4j
Image = (0.12+0.16j)
```

[16]:
```
graphWithOriginVectors(z3, w3)
```



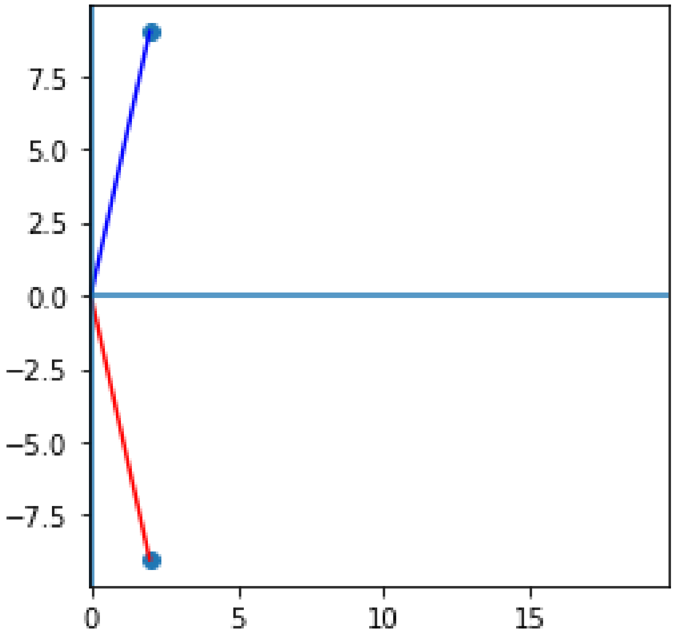
10.4 Reflection

[7]:
```
def reflect(z, option):
    """
    option = "re" => reflect along the real axis
    option = "im" => reflect along the imaginary axis
    """
    try:
        return {
        "im": z.conjugate(),
        "re": complex(-z.real, z.imag)}[option]
    except: return
```

[10]:
```
z4 = inputComplex("Preimage = ")
option = input("Option = ")
w4 = reflect(z4, option)
print("Image =", w4)
```

```
Preimage = 2-9j
Option = im
Image = (2+9j)
```

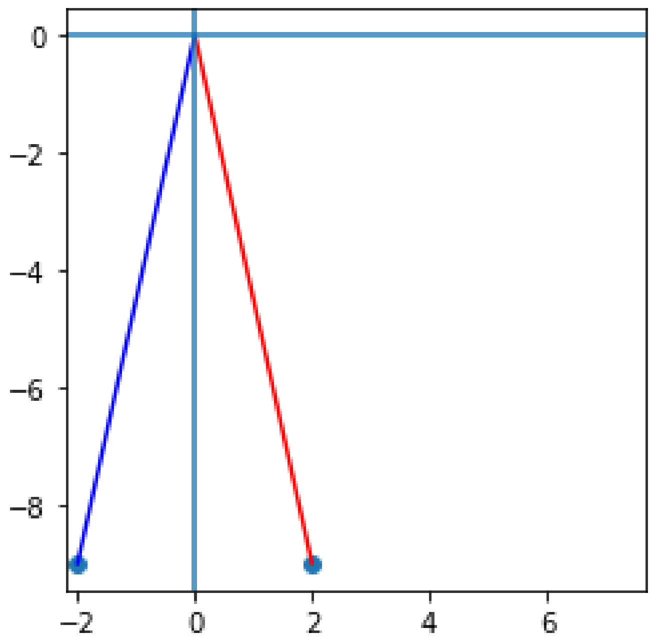
[13]:
```
graphWithOriginVectors(z4, w4)
```



[14]:
```
z4 = inputComplex("Preimage = ")
option = input("Option = ")
w4 = reflect(z4, option)
print("Image =", w4)
```

```
Preimage = 2-9j
Option = re
Image = (-2-9j)
```

[15]:
```
graphWithOriginVectors(z4, w4)
```



11 Conformity of transformations

11.1 Introduction

A transformation is said to be conformal if it preserves the orientation and magnitude of the angle of intersubsection between any two curves. In other words, consider curves c_1 and c_2 that are transformed to c'_1 and c'_2 respectively, under some transformation $w = f(z)$. Then, this transformation is conformal if and only if the angle of intersubsection at a certain direction between c_1 and c_2 is the same as the angle of intersubsection at the same direction between c'_1 and c'_2.

According to a result, at each point where the function $f(z)$ is analytic and $f'(z) \neq 0$, the mapping between z and $f(z)$ is conformal. Using this result, we will check if a given transformation is conformal within a certain range. Note that to test if a function f is analytic for a given point $z = x + iy,\ x, y \in R$, then $\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y}$ and $\frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x}$ i.e. the Cauchy-Riemann equations are satisfied.

To check if the derivative of a transformation $f(z)$ is ever zero, we can solve for the equation $f'(z) = 0$. If the number of solutions is zero, we will know that the derivative of the transformation is never zero. If the number of solutions is greater than 0, we can return the solutions so that the user can identify at what point(s) is the transformation non-conformal.

11.1.1 Notes

We define the following notations for this file...

- $z = x + iy,\ x, y \in R$ is a complex number in the domain of the transformation
- $w = f(z) = u + iv,\ u, v \in R$ is a complex number in the range of the transformation

Note that $f(z)$ is essentially a complex function, and like all complex functions, maps one set of complex numbers to another set of complex numbers.

11.2 Checking if the transformation is analytic

[2]:
```
from sympy import Symbol, I, sympify
def isAnalytic(u, v, val=[]): # u and v are strings of expressions␣
↪containing x and y
    """
    Notes:
    1.
    z = u + iv, where u and v are real valued functions.

    2.
    val is a list of two lists.
    The 1st list is the range of the real part.
    The 2nd list is the range of the imaginary part.
    """
    x, y, pd = Symbol('x'), Symbol('y'), []
    # Obtaining partial derivatives
    pd.append(u.diff(x))
    pd.append(v.diff(y))
    pd.append(u.diff(y))
    pd.append(v.diff(x))

    # If we need to check at given points...
    if len(val) == 2:
        # Setting last index as minimum length of the given sublists
        top = min(len(val[0]), len(val[1]))

        # Iterating through each set of real and imaginary parts given
        for i in range(0, top):
            re, im = val[0][i], val[1][i]
            tmp = []
            for j in range(0, 4):
                tmp.append(pd[j].subs({x:re, y:im}))
```

```
            # If not analytic at even one point, return false
            if not (tmp[0]==tmp[1] and tmp[2]==-tmp[3]): return False
        return True

    # If we need to check in general...
    return pd[0]==pd[1] and pd[2]==-pd[3]
```

Testing above function…

[3]:
```
print("\nEXAMPLE 1")
u = sympify("x^2-y^2")
v = sympify("2*x*y")
# NOTE: sympify recognizes both '^' and '**' for power operator.
print("u:", u)
print("v:", v)
print("Analytic in general?", isAnalytic(u, v))
print("Analytic at (3, -3)?", isAnalytic(u, v, [[3], [-3]]))
#------------------------
print("\nEXAMPLE 2")
print("(Analytic at a point)")
u = sympify("x^3")
v = sympify("y^3")
print("u:", u)
print("v:", v)
print("Analytic in general?", isAnalytic(u, v))
print("Analytic at (0, 0)?", isAnalytic(u, v, [[0], [0]]))
```

```
EXAMPLE 1
u: x**2 - y**2
v: 2*x*y
Analytic in general? True
Analytic at (3, -3)? True

EXAMPLE 2
(Analytic at a point)
u: x**3
v: y**3
Analytic in general? False
Analytic at (0, 0)? True
```

11.3 Checking the derivative of transformation i.e. $f'(z)$

To obtain the derivative of the complex function (call it $f(z)$) using the real part alone, we use the equation $f'(z) = \frac{\partial u}{\partial x} + i\frac{\partial u}{\partial y}$

[4]:
```
from sympy import Symbol, I, sympify, solve, Eq
def derivative(u):
    # u is the real part of a complex function f(z)
    return u.diff(Symbol('x'))-I*u.diff(Symbol('y'))

def willDerivativeBeNonZero(u, val=[]):
    # u is the real part of a complex function f(z)
    """
    Notes (as before):
    1.
    z = u + iv, where u and v are real valued functions.

    2.
    val is a list of two lists.
    The 1st list is the range of the real part.
    The 2nd list is the range of the imaginary part.
    """
    # Defining symbols
    x, y = Symbol('x'), Symbol('y')

    # Obtaining f'(z)
    df_dz = derivative(u)

    # If we need to check at given points...
    if len(val) == 2:
        # Setting last index as minimum length of the given sublists
        top = min(len(val[0]), len(val[1]))

        # Iterating through each set of real and imaginary parts given
        for i in range(0, top):
            re, im = val[0][i], val[1][i]
            tmp = df_dz.subs({x:re, y:im})
            if tmp == 0: return False
        return True

    # If we need to check in general...
    if df_dz == 0: return False
```

```
    # Checking if df/dz = 0 has any solutions i.e. if derivative is ever␣
↪zero
    if len(solve(Eq(df_dz, 0))) == 0: return True
    return False
```

Testing the above functions…

[5]:
```
derivative(sympify("x**2-y**2"))
```

[5]:

$$2x + 2iy$$

[6]:
```
derivative(sympify("log(sqrt(x**2 + y**2))"))
```

[6]:

$$\frac{x}{x^2 + y^2} - \frac{iy}{x^2 + y^2}$$

[7]:
```
print("Will derivative of x**2-y**2 be non-zero...")
print("For all?")
print(willDerivativeBeNonZero(sympify("x**2-y**2")))
print("For the point (0, 3)?")
print(willDerivativeBeNonZero(sympify("x**2-y**2"), [[0], [3]]))
print("For the point (0, 0)?")
print(willDerivativeBeNonZero(sympify("x**2-y**2"), [[0], [0]]))
print("For both points (0, 1) and (1, -2)?")
print(willDerivativeBeNonZero(sympify("x**2-y**2"), [[0, 1], [1, -2]]))
```

```
Will derivative of x**2-y**2 be non-zero...
For all?
False
For the point (0, 3)?
True
For the point (0, 0)?
False
For both points (0, 1) and (1, -2)?
True
```

11.4 Checking if the transformation is conformal

We could merge the above two functionalities into a single function…

[12]:
```
from sympy import Symbol, I, sympify
def isConformal(u, v, val=[]): # u and v are strings of expressions␣
↪containing x and y
    """
    Notes:
```

```
    1.
    z = u + iv, where u and v are real valued functions.

    2.
    val is a list of two lists.
    The 1st list is the range of the real part.
    The 2nd list is the range of the imaginary part.
    """
    x, y, pd = Symbol('x'), Symbol('y'), []
    # Obtaining partial derivatives
    pd.append(u.diff(x))
    pd.append(v.diff(y))
    pd.append(u.diff(y))
    pd.append(v.diff(x))

    # Obtaining complete derivative
    df_dz = u.diff(x)-I*u.diff(y)

    # If we need to check at given points...
    if len(val) == 2:
        # Setting last index as minimum length of the given sublists
        top = min(len(val[0]), len(val[1]))

        # Iterating through each set of real and imaginary parts given
        for i in range(0, top):
            re, im = val[0][i], val[1][i]

            # If f'(z) = 0, transformation f(z) is not conformal
            if df_dz.subs({x:re, y:im}) == 0: return False

            # If f'(z) =/= 0, checking if analytic
            tmp = []
            for j in range(0, 4):
                tmp.append(pd[j].subs({x:re, y:im}))

            # If not analytic at even one point, return false
            if not (tmp[0]==tmp[1] and tmp[2]==-tmp[3]): return False
        return True
    #------------------------
    # If we need to check in general...
    #____________
    # Handling the obvious cases...
```

```
    if df_dz == 0: return False
    if not (pd[0]==pd[1] and pd[2]==-pd[3]): return False
    #____________
    # Handling the less obvious cases...
    # Checking if df/dz = 0 has any solutions i.e. if derivative is ever␣
↪zero
    solutions = solve(Eq(df_dz, 0))
    if len(solutions) == 0: return True
    # The solutions for x and y show where the transformation is␣
↪non-conformal
    return solutions
```

Testing the above function...

[13]:
```
u = sympify("x**2-y**2")
v = sympify("2*x*y")
isConformal(u, v, [])
```

[13]: [{x: -I*y}]

Hence, we can see that the transformation is non-conformal when $x = -iy$. Since x and y are real numbers, this equation can only be true if $x = y = 0$. Hence, the transformation is non-conformal at $(0, 0)$.

11.5 Input functions

[14]:
```
from numpy import linspace
def getBound(prompt, default):
    s, flag = input(prompt).strip(), 0
    # Checking for * in the end
    if s[-1] == "*": s, flag = s[:-1], 1
    try: s = float(s)
    except:
        print("Couldn't convert " + s + " to float.")
        print("Defaulting to " + str(default) + "...\n")
        s = default
    return [s, flag]
def inputInterval():
    print("(Append * to the bound value to exclude from the interval)")
    flag = 0
    lower = getBound("Lower bound: ", 0)
    upper = getBound("Upper bound: ", 0)
    # If lower bounds exceeds upper bound, switch
```

```
    if upper[0] < lower[0]:
        tmp = upper
        upper = lower
        lower = tmp

    # Obtaining divisions for the interval
    divs = input("Divisions: ")
    try: divs = abs(int(divs))
    except:
        print("Couldn't convert " + divs + " to int.")
        print("Defaulting to 0...")
        divs = 10

    # If inclusive, we get flag = 0,
    # and add 0 to the bound.
    # If exclusive, we get flag = 1,
    # and add (upper[0]-lower[0])/divs to the bound.
    lower[0] = lower[0] + lower[1]*float(upper[0]-lower[0])/divs
    upper[0] = upper[0] + upper[1]*float(upper[0]-lower[0])/divs

    # Obtaining interval
    return linspace(lower[0], upper[0], divs)

def inputComplexFunction():
    try:
        u = sympify(input("Re(f(z)): "))
        v = sympify(input("Im(f(z)): "))
    except:
        print("Invalid expressions!")
        return

    X, Y = [], []
    option = input("Give specifics? (y/n) ")
    if option == "y":
        print("\nInput desired real part range:")
        X = inputInterval()
        print("\nInput desired imaginary part range:")
        Y = inputInterval()

    D = [X, Y]
    if len(X) == 0 or len(Y) == 0: D = []
    return [u, v, D]
```

Testing the above functions. . .

[240]:
```
inputComplexFunction()
```

```
Re(f(z)): y*x+23
Im(f(z)): x-y
Give specifics? (y/n) n
```

[240]:
```
[x*y + 23, x - y, [[], []]]
```

[241]:
```
inputComplexFunction()
```

```
Re(f(z)): 2**x
Im(f(z)): y*x
Give specifics? (y/n) n
```

[241]:
```
[2**x, x*y, [[], []]]
```

[242]:
```
inputComplexFunction()
```

```
Re(f(z)): x**2
Im(f(z)): y-x
Give specifics? (y/n) y

Input desired real part range:
(Append * to the bound value to exclude from the interval)
Lower bound: 1
Upper bound: 3
Divisions: 3

Input desired imaginary part range:
(Append * to the bound value to exclude from the interval)
Lower bound: 2*
Upper bound: 4
Divisions: 5
```

[242]:
```
[x**2, -x + y, [array([1., 2., 3.]), array([2.4, 2.8, 3.2, 3.6, 4. ])]]
```

11.6 Application

We will use the following functions and their dependencies. . .

- `inputComplexFunction`
- `isConformal`

[28]:
```
def checkConformity():
    s = isConformal(*inputComplexFunction())
    if s == True:
        print("Inputted mapping is conformal for the given range.")
    else:
        print("Inputted mapping is not conformal for the given range.")
        print("Reasons for non conformity:")
        if s == False:
            print("- Function is not analytic for the given range.")
        else:
            print("- Function is non-conformal given following...")
            print("  ", s)
```

[23]:
```
checkConformity()
```

```
Re(f(z)): x
Im(f(z)): y
Give specifics? (y/n) n
Inputted mapping is conformal for the given range.
```

[26]:
```
checkConformity()
```

```
Re(f(z)): x**2
Im(f(z)): y**3
Give specifics? (y/n) n
Inputted mapping is not conformal for the given range.
Reasons for non conformity:
- Function is not analytic for the given range.
```

[29]:
```
checkConformity()
```

```
Re(f(z)): x**2-y**2
Im(f(z)): 2*x*y
Give specifics? (y/n) n
Inputted mapping is not conformal for the given range.
Reasons for non conformity:
- Function is non-conformal given following...
  [{x: -I*y}]
```

[30]:
```
checkConformity()
```

```
Re(f(z)): x**2-y**2
Im(f(z)): 2*x*y
Give specifics? (y/n) y
```

```
Input desired real part range:
(Append * to the bound value to exclude from the interval)
Lower bound: -1
Upper bound: 30
Divisions: 500

Input desired imaginary part range:
(Append * to the bound value to exclude from the interval)
Lower bound: 3
Upper bound: 60
Divisions: 500
Inputted mapping is conformal for the given range.
```