Example 1

Create a python class using `sympy` library that solves the following problem:
Let us consider the following equation, with unknown z and parameter φ:

$$z^2 - \frac{2}{\cos\varphi} z + \frac{5}{\cos^2\varphi} - 4 = 0$$

with $-\frac{\pi}{2} < \varphi < \frac{\pi}{2}$. Solve this equation for z.

In [ ]:
```python
import sympy as sp


class Solver:
    def __init__(self):
        self.phi = sp.symbols("phi")
        self.z = sp.symbols("z")
        self.equation = (
            self.z**2
            - (2 / sp.cos(self.phi)) * self.z
            + (5 / sp.cos(self.phi) ** 2)
            - 4
        )
        self.constrain = sp.And(-sp.pi / 2 < self.phi, self.phi < sp.pi / 2)
        self.solutions = self.solve_equation()

    def solve_equation(self):
        solutions = sp.solveset(self.equation, self.z)
        return solutions.refine(self.constrain).simplify()

    def get_solutions(self):
        for sol in self.solutions:
            print(sol)


solver = Solver()
solver.solutions
```

$$\left\{\frac{1 - 2\sqrt{-\sin^2(\phi)}}{\cos(\phi)}, \frac{2\sqrt{-\sin^2(\phi)} + 1}{\cos(\phi)}\right\}$$

Example 2

Write a Bisection algorithm class that can be used to find the root of a function. The `__init__` method should take in the function, the tolerance, and the maximum number of iterations. The class should have a `find_root` method that takes in the lower and upper bounds of the interval to search for the root. The `find_root` method should return the root of the function if it is found within the tolerance, otherwise it should print a message saying that the root was not found within the maximum number of iterations and return `None`.

In [ ]:
```python
class BisectionMethod:
    def __init__(self, function, tolerance=1e-7, max_iterations=100):
        """
        Initializes the BisectionMethod class.
        """
        self.function = function
```

```python
        self.tolerance = tolerance
        self.max_iterations = max_iterations

    def find_root(self, a, b):
        """
        Applies the Bisection Method to find the root of the function.
        """
        if self.function(a) * self.function(b) >= 0:
            print(f"Functiond does not have a root between {a} and {b}")
            return None

        for iteration in range(self.max_iterations):
            c = (a + b) / 2  # Midpoint
            f_c = self.function(c)

            # Check if the midpoint is a root or if the tolerance is met
            if abs(f_c) < self.tolerance or (b - a) / 2 < self.tolerance:
                return c

            # Decide the side to repeat the process
            if self.function(a) * f_c < 0:
                b = c
            else:
                a = c

        print(f"The method did not converge after {self.max_iterations} iterations.")
        return None


def example_function(x):
    return x * (x - 2) * (x + 4)


bisection = BisectionMethod(example_function)
# print(help(bisection))
root = bisection.find_root(1, 2.5)
print(f"The approximate root is: {root:.2f}")
print(f"The function value at {root:.2f} is: {bisection.function(root):.3e}")
```

```
The approximate root is: 2.00
The function value at 2.00 is: 3.576e-07
```

Example 3

The trapezoidal rule is a numerical method to evaluate definite integrals. It approximates the integral of a function $f(x)$ by the sum of the areas of trapezoids formed by the function and the x-axis between two points a and b. The formula for the general case is:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} \frac{f(x_{i-1}) + f(x_i)}{2} \Delta x,$$

where $x_i = a + i\Delta x$ and $\Delta x = \dfrac{b-a}{n}$. Or we can write it as:

$$\int_a^b f(x)dx \approx \frac{f(a) + f(b)}{2} \Delta x + \sum_{i=1}^{n-1} f(a + i\Delta x)\Delta x.$$

Write a class `TrapezoidalMethod` that takes in a function $f(x)$, the exact integral of the function, and the interval $[a, b]$ as arguments. The class should have

- a method `__init__` that initializes the function and the interval
- a method `approximate_integral` that returns the approximate integral of the function using

the trapezoidal rule

- a method `error` that returns the error in the approximation of the integral
- a method `relative_error` that returns the relative error in the approximation of the integral

In [ ]:
```
def f(x):
    return x**2 - 2


def exact_integral(x):
    return x**3 / 3 - 2 * x


class TrapezoidalMethod:
    def __init__(self, function, exact_integral, a, b, n=10):
        self.function = function
        self.exact_integral = exact_integral
        self.a = a
        self.b = b
        self.n = n

    def approximate_integral(self):
        delta_x = (self.b - self.a) / self.n
        summation = 0.5 * (self.function(self.a) + self.function(self.b))

        for i in range(1, self.n):
            summation += self.function(self.a + i * delta_x)

        return delta_x * summation

    def error(self):
        exact = self.exact_integral(self.b) - self.exact_integral(self.a)
        return abs(exact - self.approximate_integral())

    def relative_error(self):
        exact = self.exact_integral(self.b) - self.exact_integral(self.a)
        return abs(exact - self.approximate_integral()) / abs(exact)


x_min = 0
x_max = 2
trapezoidal = TrapezoidalMethod(f, exact_integral, x_min, x_max, 1000)
integral = trapezoidal.approximate_integral()
print(f"The approximate integral between {x_min} and {x_max} is: {integral:.2f}")
print(f"The error is: {trapezoidal.error():.3e}")
print(f"The relative error is: {trapezoidal.relative_error():.3e}")
```

```
The approximate integral between 0 and 2 is: -1.33
The error is: 1.333e-06
The relative error is: 1.000e-06
```

Example 4

The Lagrange multiplier method is a mathematical technique for finding the local maxima and minima of a function subject to equality constraints. The method involves forming the Lagrange function, which is the function to be optimized along with the constraints. The method of Lagrange multipliers states that the gradient of the Lagrange function at the optimal point is proportional to the gradient of the constraint function.

Create a class `LagrangeMultiplierOptimizer` that takes in a function $f(x, y)$ and a constraints $g(x, y) = 0$ as arguments.

- The class should calculate the Lagrange function $L(x, y, \lambda) = f(x, y) + \lambda g(x, y)$.

- The class should have a method `optimize` that finds the minimum or maximum of the function $f(x, y)$ subject to the constraint $g(x, y) = 0$ using the method of Lagrange multipliers. The method should return the optimal values of x, y and λ parameters that satisfy the constraint and optimize the function.