In [1]:
```python
import numpy as np
```

In [2]:
```python
import matplotlib.pyplot as plt
```

Question 1

Consider the function $f(x) = x^2$. Find its condition number and discuss its sensitivity. Repeat the exercise for the function $g(x) = \sin x$

The condition number for a function $f(x)$ can be approximated by:

$$\text{condition number} = \left| \frac{x f'(x)}{f(x)} \right|$$

Using this, we can get the following for the three functions:

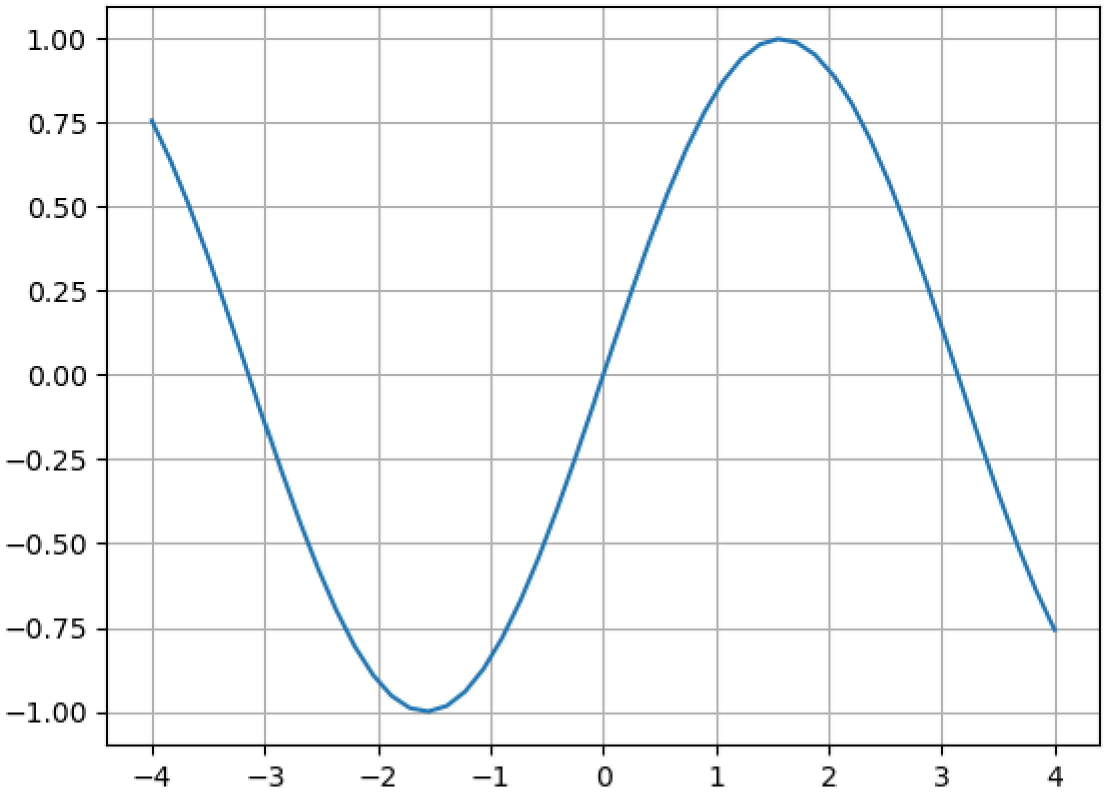
$$\begin{aligned}\text{condition number}_f &= \left| \frac{x f'(x)}{f(x)} \right| \\ &= \left| \frac{x \times 2x}{x^2} \right| \\ &= \left| \frac{2x^2}{x^2} \right| \\ &= 2\end{aligned}$$

Since the condition number for f is greater than 1, we can determine that it is ill-conditioned and thus sensitive.

$$\begin{aligned}\text{CN}_g &= \left| \frac{x g'(x)}{g(x)} \right| \\ &= \left| \frac{x \cos(x)}{\sin(x)} \right| \\ &= |x \cot(x)|\end{aligned}$$

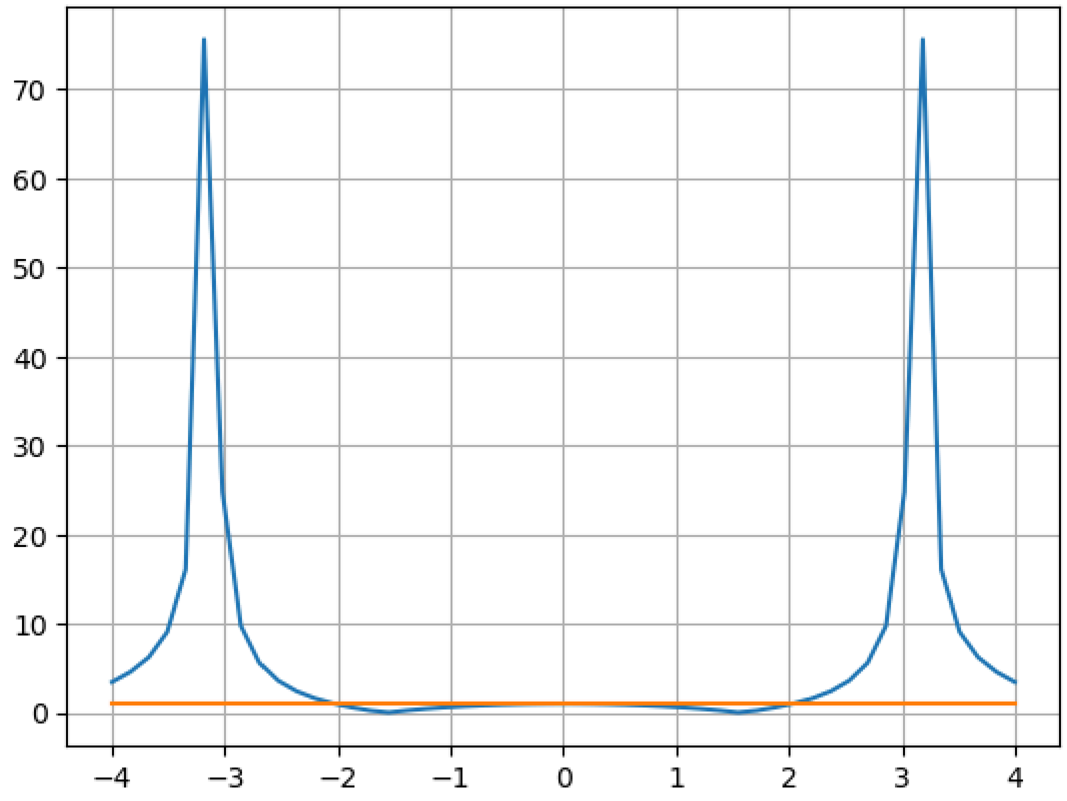
We have to analyse the result for g a little bit more carefully. The expression for the condition number of g is infinite for multiples of π. Thus, the problem is ill-conditioned around π. To see this in practice consider $\sin(3.14) \approx 1.59 \times 10^{-3}$, and $\sin(3.141) \approx 5.92 \times 10^{-4}$. The relative change in the output is much larger than the relative change in the input. This can be seen in the plots below, first the function and then the condition number...

In [3]:
```python
x = np.linspace(-4, 4)
plt.plot(x,np.sin(x))
plt.grid()
```



```
In [4]: def condition_number(x):
            return np.abs(x*np.cos(x)/np.sin(x))
```

```
In [5]: plt.plot(x,condition_number(x))
        plt.plot(x, np.ones(len(x)))
        plt.grid()
```



Question 2

Provide the LU decomposition of:

$$A = \begin{pmatrix} 1 & 3 & -2 \\ 1 & 5 & 3 \\ 2 & 12 & 10 \end{pmatrix}$$

Answer:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 3 & 1 \end{pmatrix}$$

$$U = \begin{pmatrix} 1 & 3 & -2 \\ 0 & 2 & 5 \\ 0 & 0 & -1 \end{pmatrix}$$

Method:

We first find a sequence of elimination matrices to produce the upper triangular form.

For column 1:

$$M_1 = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix}$$

We observe that:

$$M_1 \cdot A = \begin{pmatrix} 1 & 3 & -2 \\ 0 & 2 & 5 \\ 0 & 6 & 14 \end{pmatrix}$$

Thus, for eliminating the second column we obtain:

$$M_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -3 & 1 \end{pmatrix}$$

This gives us U.

$$U = M_2 M_1 A = \begin{pmatrix} 1 & 3 & -2 \\ 0 & 2 & 5 \\ 0 & 0 & -1 \end{pmatrix}$$

We can find matrices L_1 and L_2, by multiplying times -1 the multipliers in each M_1 and M_2 respectively. $L = L_1 \cdot L_2$.

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix} \quad L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{pmatrix} \quad L = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 3 & 1 \end{pmatrix}$$

It's easy to verify that $L \cdot U = A$.

Question 3

Implement a function to compute the LU decomposition of any given square matrix A.

In [6]:
```python
def forward_elimination_matrix(A, p):
    #p is the column index I want to "zero out"
    pivot = A[p,p]
    M = np.eye(len(A)) #start with the identity matrix
    for row_index in range(p+1, len(A)):
        M[row_index, p] = -A[row_index, p]/pivot
    return M
```

In [7]:
```python
def LU(A):
    # let us not mess with A
    U = np.copy(A) # remember U is M3M2M1A
    L = np.eye(len(A)) # just a product of lower trig matrices

    # go through all columns
    for i in range(len(A)):
        M = forward_elimination_matrix(U, i)
        U = np.dot(M, U)
        L[i+1:, i] = -M[i+1:, i]

    return L, U
```

In [8]:
```python
A = np.array([[1, 2, 2],[4, 4, 2], [4, 6, 4]])
```

In [9]:
```python
L, U = LU(A)
```

In [10]:
```python
np.dot(L, U)
```

Out[10]:
```
array([[1., 2., 2.],
       [4., 4., 2.],
       [4., 6., 4.]])
```

Question 4

Solve the following system of equations:

$$\begin{aligned} 0.0001x + y &= 1 \\ x + y &= 2 \end{aligned}$$

First solve this system using Gaussian Elimination or LU decomposition then solve it again using partial pivoting. In both cases, use a precision of 3 significant figures.

Without pivoting we do the operation: $R_2 = R_2 - 1000 \times R_1$

$$\begin{matrix} 0.0001 & 1 & 1 \\ 1 & 1 & 2 \end{matrix} \sim \begin{matrix} 0.0001 & 1 & 1 \\ 0 & \backslash\text{cancel}-9999_{-10^5} & \backslash\text{cancel}-9998_{-10^5} \end{matrix}$$

When we apply the precision rule, we can notice that the two stiked out values get rounded to the subscripted values. Reading off the solutions now, it is clear from row 2 that $y = 1$. Substituting this in row 1, we get:

$$\begin{aligned} 0.0001x + 1 &= 1 \\ \implies x &= 0 \end{aligned}$$

And so the solution is $x = 0$, $y = 1$, which is clearly incorrect when we observe the second equation given.

Using pivoting, we first swap rows 1 and 2 so that the largest value in column 1 is used as a pivot. We then do the operation: $R_2 = R_2 - 0.0001 \times R_1$

$$\begin{matrix} 1 & 1 & 2 \\ 0.0001 & 1 & 1 \end{matrix} \sim \begin{matrix} 1 & 1 & 2 \\ 0 & \backslash\text{cancel}0.9999_1 & \backslash\text{cancel}0.9998_1 \end{matrix}$$

From row 2 we have $y = 1$. Substituting this into row 1, we get:

$$\begin{aligned} x + 1 &= 2 \\ \implies x &= 1 \end{aligned}$$

And so the solution is $x = 1$, $y = 1$, which is a lot more suitable.

Question 5

Implement a function to compute the PLU decomposition of any given square matrix A using pivoting.

In [13]:
```python
def pivot(A, r1, r2):
    P = np.eye(len(A))
    P[r1, r1], P[r2, r2] = 0, 0
    P[r1, r2], P[r2, r1] = 1, 1
    return P

def max_row(A, r):
    return np.argmax(A[r:, r])+r
    # return np.argmax(A[:, r:])+r
```

In [14]:
```python
def LUP(A):
    # let us not mess with A
    U = np.copy(A) # remember U is M3M2M1A
    L = np.eye(len(A)) # just a product of lower trig matrices
    P = np.eye(len(A))

    for i in range(len(A)):
        p = max_row(U, i)
        Pi = pivot(U, r1=i, r2=p)
        P = np.dot(Pi, P)
        U = np.dot(Pi, U)
        M = forward_elimination_matrix(U, p=i)
        U = np.dot(M, U)
        # L[i+1:, i] = -M[i+1:, i]
        Li = np.linalg.inv(M)
        L = np.dot(L, Pi.T)
        L = np.dot(L, Li)
    L = np.dot(P, L)
    return L, U, P
```

In [15]:
```python
L, U, P = LUP(A)
L, U, P
```

```
Out[15]: (array([[1.  , 0.  , 0.  ],
                 [1.  , 1.  , 0.  ],
                 [0.25, 0.5 , 1.  ]]),
          array([[4. , 4. , 2. ],
                 [0. , 2. , 2. ],
                 [0. , 0. , 0.5]]),
          array([[0., 1., 0.],
                 [0., 0., 1.],
                 [1., 0., 0.]]))
```