# Direct Methods for the solution of Linear Systems

In [ ]:
```
import numpy as np
import scipy
import matplotlib.pyplot as plt
```

## Step 1

• Computes the right-hand side of the linear system b = A * x_true.

• Computes the condition number in 2-norm of the matrix A. It is ill-conditioned? What if we use the ∞-norm instead of the 2-norm?

• Solves the linear system Ax = b with the function np.linalg.solve().

• Computes the relative error between the solution computed before and the true solution xtrue.

In [ ]:
```
def solveSystem(A):
    # defintion of b = A@Xtrue
    x_true = np.ones((A.shape[0],))
    b = A @ x_true
    # b will be approximated
    x_sol = np.linalg.solve(A, b)
    # relative error
    error_rel = np.linalg.norm(x_true - x_sol, 2) / np.linalg.norm(x_true, 2)

    return x_sol,error_rel
```

In [ ]:
```
def evaluate(matrixes):
    errors = []
    conds_2 = [] # condition number using 2-norm
    conds_inf = [] # using infinite-normm


    for A in matrixes:
        # condition numbers
        norm_2 = np.linalg.cond(A, 2)
        norm_inf = np.linalg.cond(A, np.inf)
        # solution and relative error
        x_sol, error_rel = solveSystem(A)
        print(f"Shape of A: {A.shape}".ljust(40) +
              f"2-norm K(A): {norm_2}".ljust(40) +
              f"| inf-norm K(A): {norm_inf}".ljust(40) +
              f"| rel error: {error_rel}")

        errors.append(error_rel)
        conds_2.append(norm_2)
        conds_inf.append(norm_inf)

    # we will try several square matrices with shape NxN
    n_values = [A.shape[0] for A in matrixes]

    plt.figure(figsize=(20, 5))
    plt.subplot(1, 2, 1)
    plt.title("Relative Errors")
    plt.plot(n_values, errors)
    plt.xlabel("n")
```

```
    plt.xticks(n_values)

    plt.subplot(1, 2, 2)
    plt.title("Conditions Numbers")
    plt.plot(n_values, conds_2, label="2-norm")
    plt.plot(n_values, conds_inf, label="inf-norm")
    plt.xlabel("n")
    plt.xticks(n_values)
    plt.legend()
    plt.show()
```

# Step 2

Test the program above with the following choices of A ∈ R n×n:

• A random matrix (created with the function np.random.rand()) with size varying with n = {10, 20, 30, . . . , 100}.

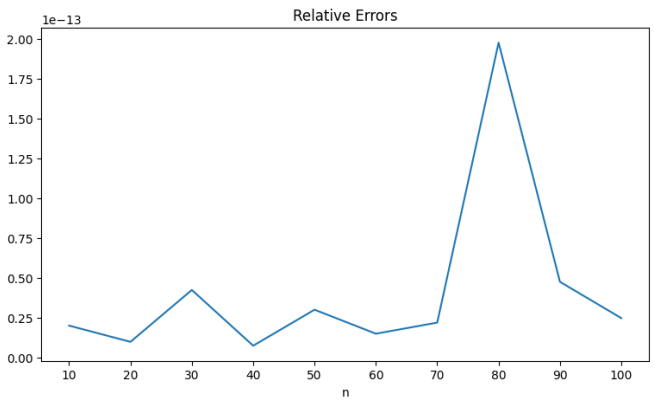
• The Vandermonde matrix (np.vander) of dimension n = {5, 10, 15, 20, 25, 30} with respect to the vector x = {1, 2, 3, . . . , n}.

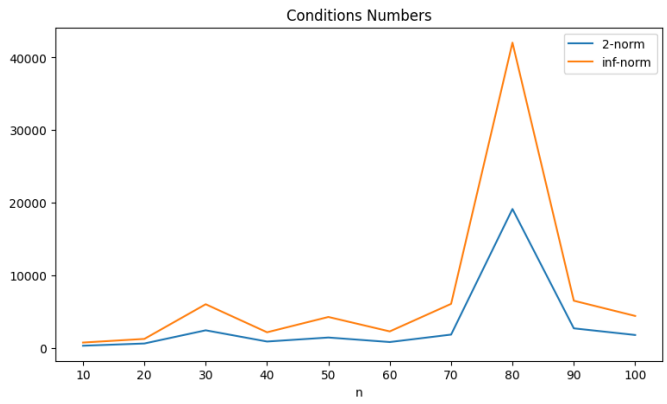
• The Hilbert matrix (scipy.linalg.hilbert) of dimension n = {4, 5, 6, . . . , 12}.

**Random matrix**

In [ ]:
```
#random.rand generates values bounded between 0 and 1
evaluate([np.random.rand(n, n) for n in range(10, 101, 10)])
```

```
Shape of A: (10, 10)                    2-norm K(A): 337.0735684308776          | inf-no
rm K(A): 772.2559294132965      | rel error: 2.0240799903805057e-14
Shape of A: (20, 20)                    2-norm K(A): 641.3635455223315          | inf-no
rm K(A): 1271.8813565958767     | rel error: 1.0087156726345954e-14
Shape of A: (30, 30)                    2-norm K(A): 2458.1974683644867         | inf-no
rm K(A): 6044.447856570096      | rel error: 4.2589313861843055e-14
Shape of A: (40, 40)                    2-norm K(A): 911.8400228046219          | inf-no
rm K(A): 2185.5437366736587     | rel error: 7.670508299835376e-15
Shape of A: (50, 50)                    2-norm K(A): 1465.030793834126          | inf-no
rm K(A): 4292.689883349334      | rel error: 3.021617135549084e-14
Shape of A: (60, 60)                    2-norm K(A): 841.5529393298826          | inf-no
rm K(A): 2306.586526806014      | rel error: 1.518847046002941e-14
Shape of A: (70, 70)                    2-norm K(A): 1867.8166028047883         | inf-no
rm K(A): 6098.336876292978      | rel error: 2.213546793384083e-14
Shape of A: (80, 80)                    2-norm K(A): 19153.756797313486         | inf-no
rm K(A): 42052.43859905905      | rel error: 1.9746112664749876e-13
Shape of A: (90, 90)                    2-norm K(A): 2727.1443011642195         | inf-no
rm K(A): 6529.477468209617      | rel error: 4.768993267350044e-14
Shape of A: (100, 100)                  2-norm K(A): 1813.3716383971177         | inf-no
rm K(A): 4434.549499275523      | rel error: 2.4946098682661474e-14
```

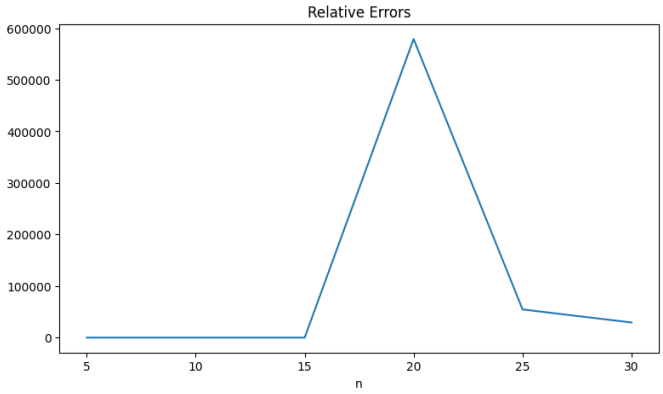


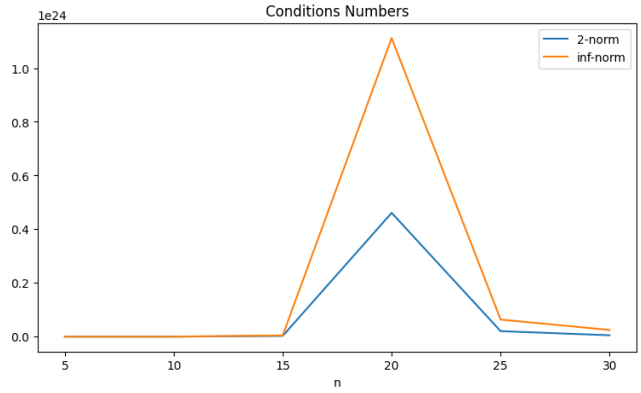



### Vandermonde matrix

In [ ]:
```
evaluate([np.vander(range(1, n+1, 1), n) for n in range(5, 31, 5)])
# for n = 20, 25 and 30 the evaluation of these parameters is influenced by the fact tha
```

```
Shape of A: (5, 5)                      2-norm K(A): 26169.687970634423         | inf-no
rm K(A): 43736.000000000524     | rel error: 1.669387470270264e-13
Shape of A: (10, 10)                    2-norm K(A): 2106257536991.8616         | inf-no
rm K(A): 3306440916902.2573     | rel error: 2.787249108403844e-07
Shape of A: (15, 15)                    2-norm K(A): 2.582409724340251e+21      | inf-no
rm K(A): 4.3640799469986476e+21 | rel error: 4.815799305987221
Shape of A: (20, 20)                    2-norm K(A): 4.6089633201547425e+23     | inf-no
rm K(A): 1.1121508501283704e+24 | rel error: 578959.9955478631
Shape of A: (25, 25)                    2-norm K(A): 2.0357813109413506e+22     | inf-no
rm K(A): 6.351061415965083e+22  | rel error: 54639.67701973694
Shape of A: (30, 30)                    2-norm K(A): 5.098965716856843e+21      | inf-no
rm K(A): 2.4729337053467543e+22 | rel error: 29337.6307920416
```

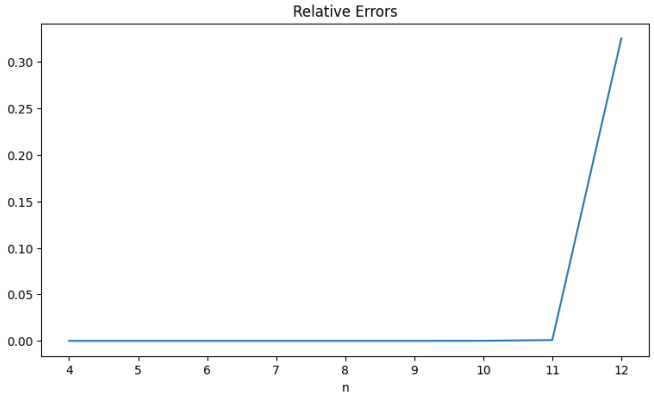


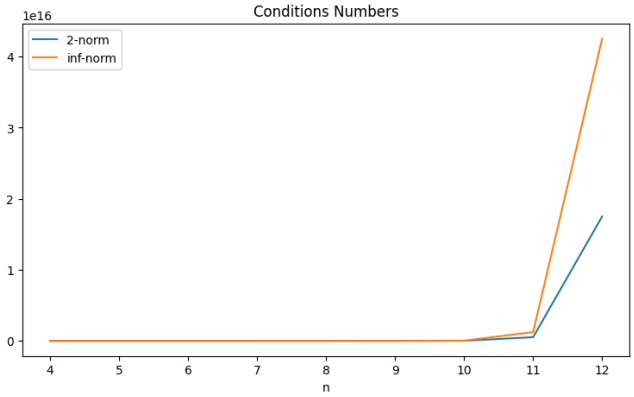




### Hilbert matrix

In [ ]:
```
evaluate([scipy.linalg.hilbert(n) for n in range(4, 13, 1)])
```

```
Shape of A: (4, 4)                      2-norm K(A): 15513.73873892924          | inf-no
rm K(A): 28375.00000000183      | rel error: 4.137409622430382e-14
Shape of A: (5, 5)                      2-norm K(A): 476607.2502425855          | inf-no
rm K(A): 943656.0000063627      | rel error: 1.6828426299227195e-12
Shape of A: (6, 6)                      2-norm K(A): 14951058.642254734         | inf-no
rm K(A): 29070279.00379062      | rel error: 1.4242437208427487e-10
Shape of A: (7, 7)                      2-norm K(A): 475367356.7446793          | inf-no
rm K(A): 985194889.577766       | rel error: 7.637452450980383e-09
Shape of A: (8, 8)                      2-norm K(A): 15257575538.060041         | inf-no
rm K(A): 33872792385.924484     | rel error: 6.124089555723088e-08
Shape of A: (9, 9)                      2-norm K(A): 493153755941.02344         | inf-no
rm K(A): 1099651994744.017      | rel error: 3.8751634185032475e-06
Shape of A: (10, 10)                    2-norm K(A): 16024416987428.36          | inf-no
rm K(A): 35356847610517.12      | rel error: 8.67039023709691e-05
Shape of A: (11, 11)                    2-norm K(A): 522270131654983.3          | inf-no
rm K(A): 1234532816741620.0     | rel error: 0.0008383287776275721
Shape of A: (12, 12)                    2-norm K(A): 1.7515952300879806e+16     | inf-no
rm K(A): 4.255399301891292e+16  | rel error: 0.3249129296869168
```




the evaluations of np.random.rand and hilbert matrix have small error values since that they handle with small values in their matrices, while the vandermonde matrix real error could be greater then what i have obteined

# Floating Point Arithmetic

## Step 1

The Machine epsilon is the distance between 1 and the next floating point number. Compute , which is defined as the smallest floating point number such that it holds: fl(1 + ) > 1 Tips: use a while structure.

In [ ]:
```
import sys
```

In [ ]:
```
eps = 1.0
while eps + 1 > 1:
    eps /= 2
eps *= 2
print("Computed eps is is = ", eps)
#smallest number for the sys
print(f"System eps is = {sys.float_info.epsilon}")
```

```
Computed eps is is =  2.220446049250313e-16
System eps is = 2.220446049250313e-16
```

In [ ]:
```
#proof that we can't divide anymore
print(1 + eps)
print(1 + eps/2)
```

```
1.0000000000000002
1.0
```

what if i use np.float128

In [ ]:
```
eps = 1.0
eps = np.float128(eps)
while eps + 1 > 1:
    eps /= 2
eps *= 2
print("Computed eps is is = ", eps)
#smallest number for the sys
print(f"System eps is = {sys.float_info.epsilon}")
```

```
Computed eps is is =  1.084202172485504434e-19
System eps is = 2.220446049250313e-16
```

# Step 2

Let's consider the sequence a_n = (1 + 1/n)^n. It is well known that: lim n → ∞ (a_n) = e, where e is the Euler costant. Choose different values for n, compute an and compare it to the real value of the Euler costant. What happens if you choose a large value of n? Guess the reason.

In [ ]:
```
import math
import numpy as np
```

In [ ]:
```
def euler(n):
    return (1 + (1/n))**n
```

In [ ]:
```
for i in range(10, 201, 10):
    approx_e = euler(i)
    difference = abs(math.e - approx_e)
    print(f"n ={i:>6}: {approx_e:.15f} (diff: {difference:.15f})")
print('\n')
for i in [1e3, 1e5, 1e10, 1e15]:
    approx_e = euler(i)
    difference = abs(math.e - approx_e)
    print(f"n ={i:>6}: {approx_e:.15f} (diff: {difference:.15f})")
```

```
n =    10: 2.593742460100002 (diff: 0.124539368359043)
n =    20: 2.653297705144422 (diff: 0.064984123314623)
n =    30: 2.674318775870303 (diff: 0.043963052588742)
n =    40: 2.685063838389963 (diff: 0.033217990069082)
n =    50: 2.691588029073608 (diff: 0.026693799385437)
n =    60: 2.695970139330216 (diff: 0.022311689128829)
n =    70: 2.699116370976185 (diff: 0.019165457482860)
n =    80: 2.701484940753327 (diff: 0.016796887705718)
n =    90: 2.703332461058186 (diff: 0.014949367400859)
n =   100: 2.704813829421528 (diff: 0.013467999037517)
n =   110: 2.706028081504754 (diff: 0.012253746954291)
n =   120: 2.707041490862244 (diff: 0.011240337596802)
n =   130: 2.707900081718078 (diff: 0.010381746740967)
n =   140: 2.708636813921145 (diff: 0.009645014537901)
n =   150: 2.709275911334851 (diff: 0.009005917124194)
n =   160: 2.709835576307815 (diff: 0.008446252151230)
n =   170: 2.710329751223865 (diff: 0.007952077235180)
n =   180: 2.710769295839407 (diff: 0.007512532619638)
n =   190: 2.711162794611157 (diff: 0.007119033847888)
n =   200: 2.711517122929317 (diff: 0.006764705529728)


n =1000.0: 2.716923932235594 (diff: 0.001357896223452)
n =100000.0: 2.718268237192297 (diff: 0.000013591266748)
n =10000000000.0: 2.718282053234788 (diff: 0.000000224775742)
n =1000000000000000.0: 3.035035206549262 (diff: 0.316753378090217)
```

In [ ]:
```
print(f"Real e: {math.e}")
print(f"float64: {euler(1e16)}") # we know that 1/1e16 = 1e-16 is too small to implement
print(f"float128: {euler(np.float128(1e16))}")
 #if we use more bit to store the addends, we can compute the euler function using the v
```

```
Real e: 2.718281828459045
float64: 1.0
float128: 2.717288214505591
```

# Step 3

Let's consider the matrices:

A = (4 2 , 1 3) , B = (4 2 , 2 1)

Compute the rank of A and B and their eigenvalues. Are A and B full-rank matrices? Can you infer some relationship between the values of the eigenvalues and the full-rank condition? Please, corroborate your deduction with other examples. Tips: Please, have a look at np.linalg.

Relation: If a matrix is full rank, then its eigenvalues are all non zero

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

In [ ]:
```
def printRankAndEigenvalues(matrix, label=""):
    rank = np.linalg.matrix_rank(matrix)
    eigenvalues, _ = np.linalg.eig(matrix)
    print(f"Matrix {label}: Rank = {rank} | Eigenvalues = {eigenvalues}")
```

In [ ]:
```
A = np.array([ [4, 2], [1, 3] ])
B = np.array([ [4, 2], [2, 1] ])

printRankAndEigenvalues(A, label="A") # is full rank
printRankAndEigenvalues(B, label="B")
```

```
Matrix A: Rank = 2 | Eigenvalues = [5. 2.]
Matrix B: Rank = 1 | Eigenvalues = [5. 0.]
```

In [ ]:
```
# Rank 2
A = np.array([ [1, 2], [0, 3] ])
printRankAndEigenvalues(A, label="A")

# Rank 1
B = np.array([ [1, 2], [0, 0] ])
printRankAndEigenvalues(B, label="B")

# Rank 0
C = np.array([ [0, 0], [0, 0] ])
printRankAndEigenvalues(C, label="C")
```

```
Matrix A: Rank = 2 | Eigenvalues = [1. 3.]
Matrix B: Rank = 1 | Eigenvalues = [1. 0.]
Matrix C: Rank = 0 | Eigenvalues = [0. 0.]
```

In [ ]:
```
# Rank 3
A = np.array([ [1, 2, 3], [0, 4, 5], [0, 0, 6] ])
printRankAndEigenvalues(A, label="A")

# Rank 2
B = np.array([ [1, 2, 3], [0, 4, 5], [0, 0, 0] ])
printRankAndEigenvalues(B, label="B")

# Rank 1
C = np.array([ [1, 2, 3], [0, 0, 0], [2, 4, 6] ])
printRankAndEigenvalues(C, label="C")

# Rank 1
D = np.array([ [1, 0, 0], [2, 0, 0], [3, 0, 0] ])
printRankAndEigenvalues(D, label="D")

# Rank 1
E = np.array([ [0, 0, 1], [0, 0, 0], [0, 12, 0] ])
printRankAndEigenvalues(E, label="E")


# Rank 0
F = np.array([ [0, 0, 0], [0, 0, 0], [0, 0, 0]
```

```
])
printRankAndEigenvalues(F, label="F")
```

```
Matrix A: Rank = 3 | Eigenvalues = [1. 4. 6.]
Matrix B: Rank = 2 | Eigenvalues = [1. 4. 0.]
Matrix C: Rank = 1 | Eigenvalues = [0. 7. 0.]
Matrix D: Rank = 1 | Eigenvalues = [0. 0. 1.]
Matrix E: Rank = 2 | Eigenvalues = [0. 0. 0.]
Matrix F: Rank = 0 | Eigenvalues = [0. 0. 0.]
```