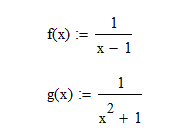
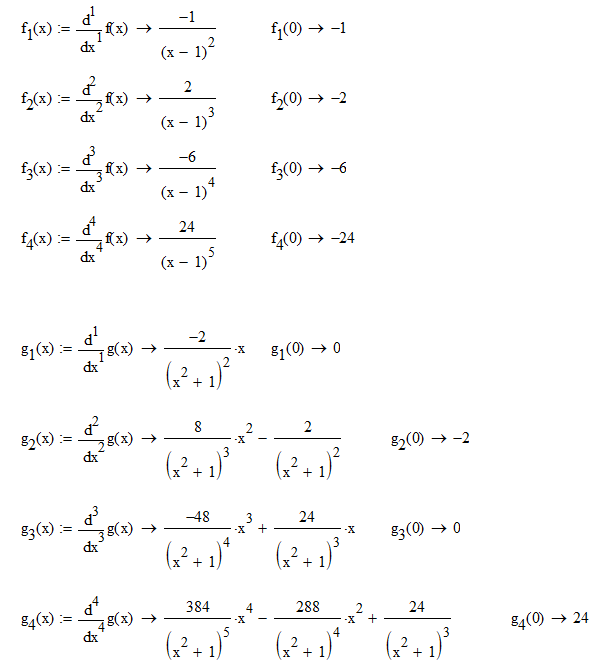
# Problem 1:

Machine generated alternative text:
and 
1. Calculate the Taylor series at the expansion point = O of the functions 
and plot the Taylor polynomials for n € {10, 20, 40} on the interval 2]. 
Hint: A partial fraction decomposition of the second function explains the bad be- 
haviour of the harmless looking function. 

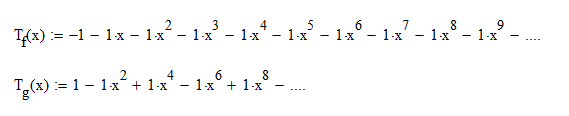
A taylor series of a given function f(x) at a point a is given as:



Writing down the first derivatives of our functions and calculating for **a=0** gives us:



Thus we can determine the following pattern for both taylor series:



The functions and their respective taylor series are implemented in Python as follows:

# Analytical function for f1

def f1(x):

    return 1.0 / (x - 1.0)

# Tatylor series for f1

# -1 - x - x^2 - x^3 - ...

def f1\_taylor(x, n):

    ret\_val = 0

    for i in range(n):

        ret\_val = ret\_val + -1.0\*x\*\*i

    return ret\_val

# Analytical function for f2

def f2(x):

    return 1.0 / (x\*\*2 + 1)

# Tatylor series for f2

# 1 - x^2 + x^4 - x^6 + x^8 ...

def f2\_taylor(x, n):

    ret\_val = 0

    for i in range(n):

        if not (i % 2): # if even

            # alternating signs

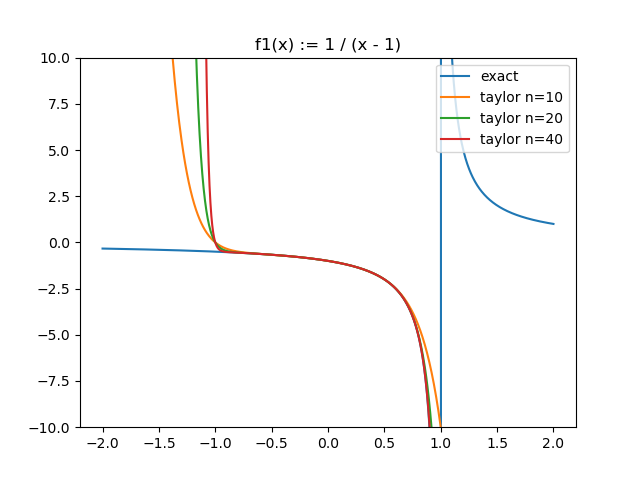
            if i % 4:

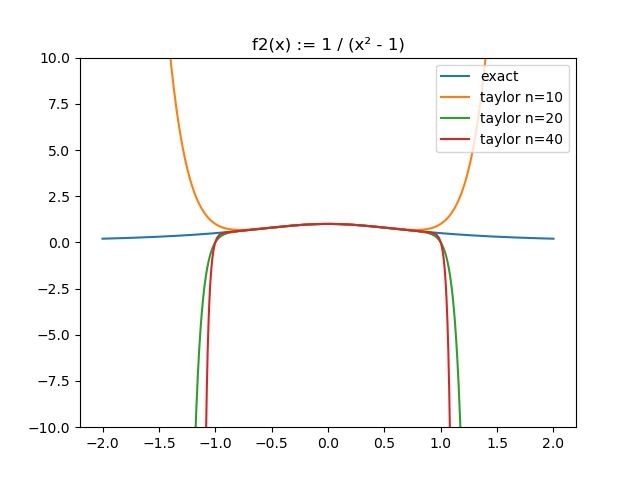
                ret\_val -= x\*\*i

            else:

                ret\_val += x\*\*i

    return ret\_val

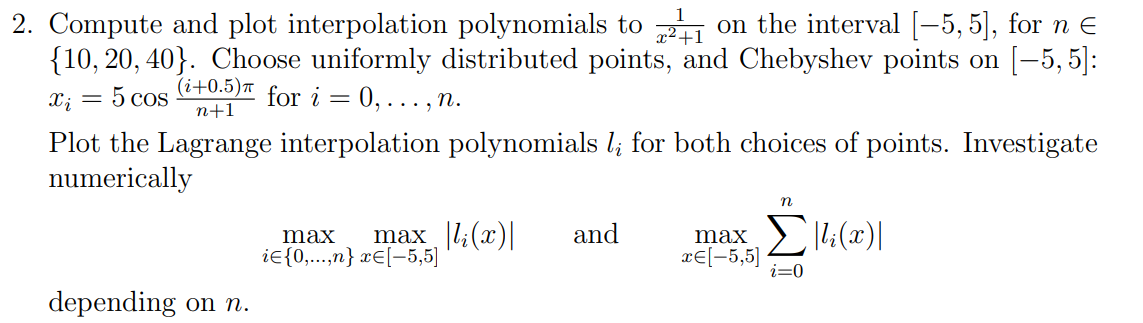
Plotting those functions with 1000 points each gives us the following results:



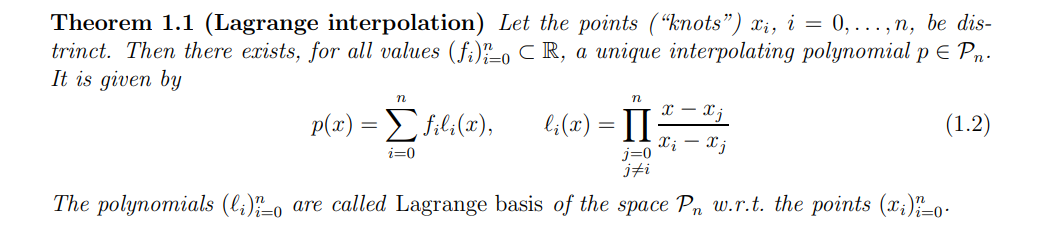
**Conclusions:**

* The taylor series can be a good approximation for non-polynomial functions within a certain area.
* The approximation is best around the point a where it is derived from. The further away we go from a the bigger the error
* If the original function is periodic or converges towards a finite value the maximum error of the Taylor approximation will always approach infinity because every polynomial of order >0 will go towards infinity
* The Taylor approximation is unable to correctly represent the pole in f1
* The higher the order of the Taylor series the better approximation. However increasing the order comes with diminishing returns. I.e. n=20 is better than n=10 but n=40 is only marginally better than n=20

# Problem 2:



The polynomial interpolation is given by:



We can implement p(x), li(x) and our given function func = (x²+1)-1 generally in Python as follows:

# l\_i

def Lagrange (pts, i, x):

    prod = 1

    for j in range(len(pts)):

        if j != i:

            prod = prod \* (x-pts[j])/(pts[i]-pts[j])

    return prod

# p(x)

def InterpolationPolynomial (fun, pts, x):

    sum = 0

    for i in range(len(pts)):

        sum = sum + fun(pts[i]) \* Lagrange(pts, i, x)

    return sum

# f(x) = 1 / (x² + 1)

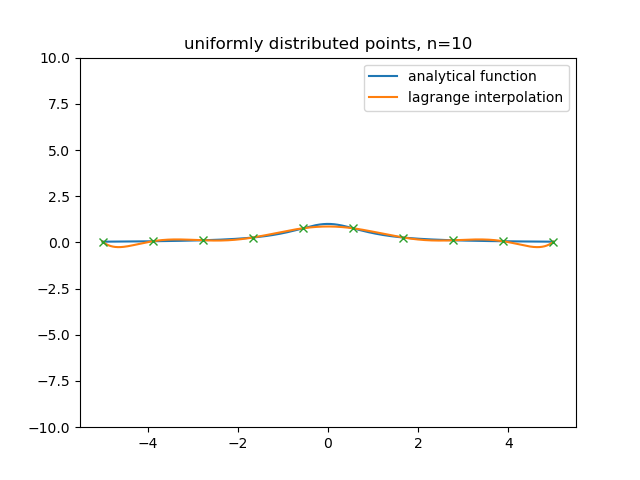
# returns a number if x is a number or a list if x is iterable object

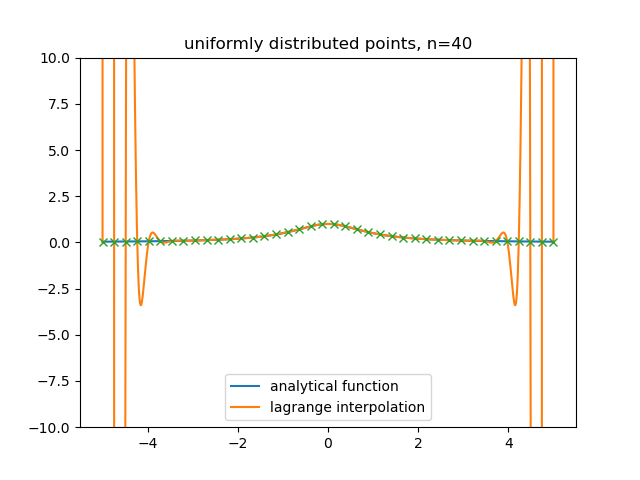
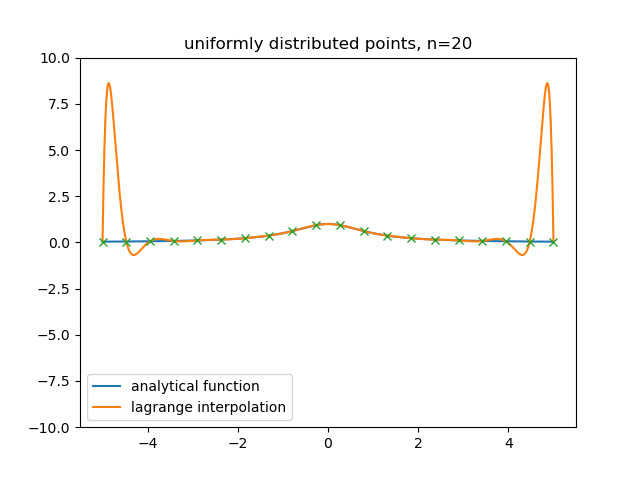
def func(x):

    if (type(x) == np.float64) or (type(x) == float): return 1.0/(1 + x\*x)

    return [1.0/(1 + x\_i\*x\_i) for x\_i in x]

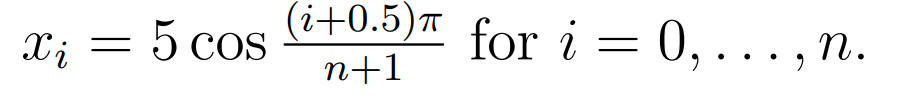
Plotting the interpolated functions with 1000 points each gives the following results:





It can be clearly seen that for higher orders the lagrange interpolation leads to some “interesting” behaviour.

One way of fixing this issue is to use Chebyshev points as described in the instructions to the problem.

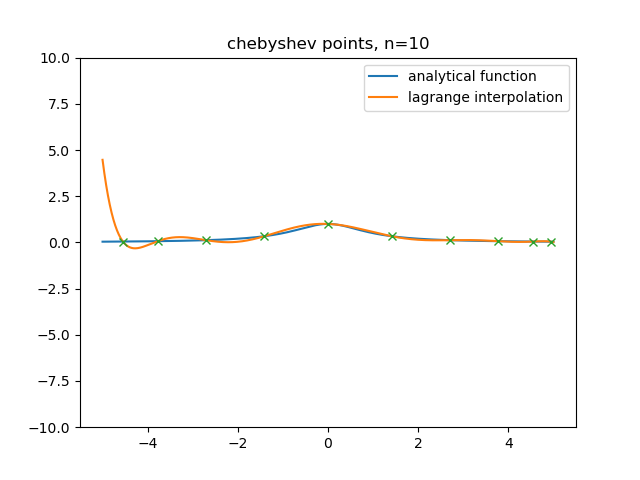


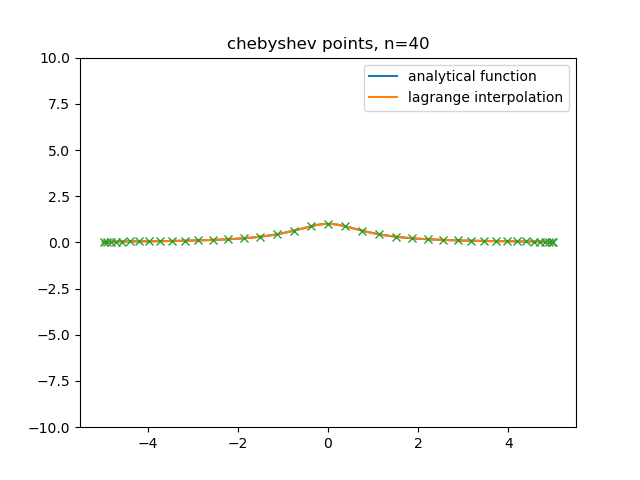
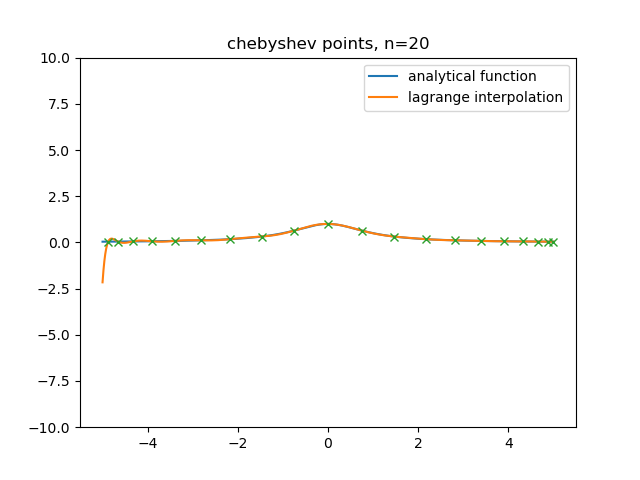
This is implemented in Python as follows:

def chebyshev\_points(start, end, count):

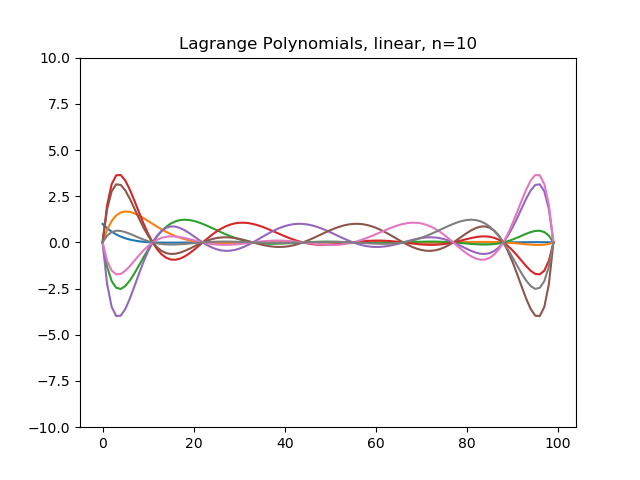
    return [5\*math.cos( (i+0.5)\*pi / (count + 1)) for i in range(count)]

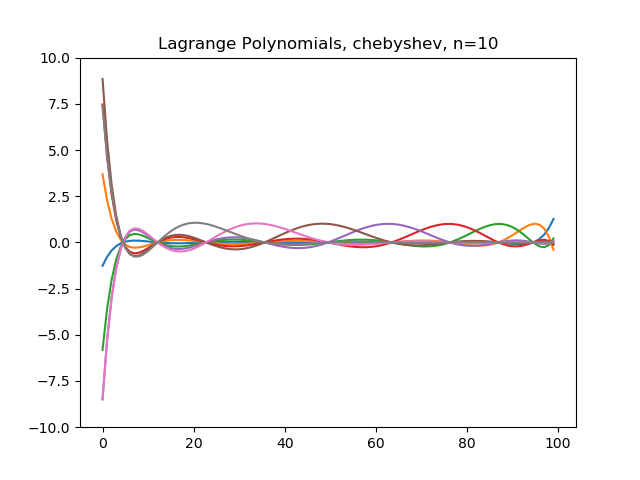
Using these points we get the following interpolations for n = 10,20,40



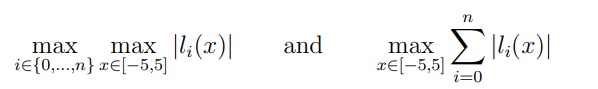


The first 8 basis functions (Lagrange polynomials) for linear distributed and chebyshev points for n=10 are as follows:





Last we are asked to plot the following maxima depending on n:



The maxima are calculated using:

# max ( max (|l\_i(x)|))

max\_vals1 = [max([max(abs(Lagrange(pts, i, x))) for i in range(n)]) for n in range(1,n\_max)]

and

# max ( sum( |l\_i(x)| ))

max\_vals2 = []

for n in range(n\_max):

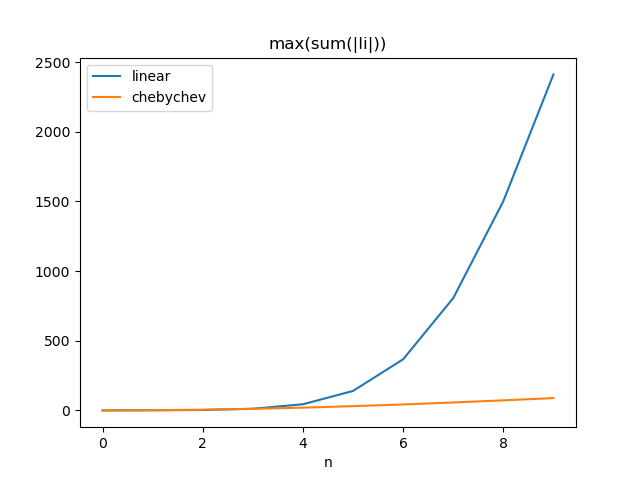
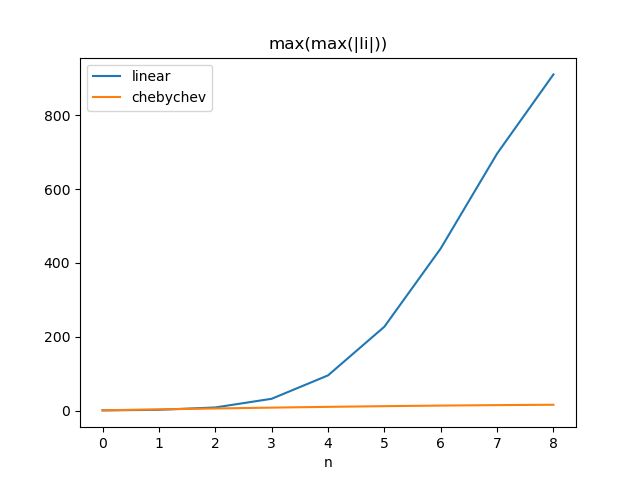
    sum = np.zeros(len(x), dtype=float)

    for i in range(n):

        sum += abs(Lagrange(pts, i, x))

    max\_vals2.append(max(sum))

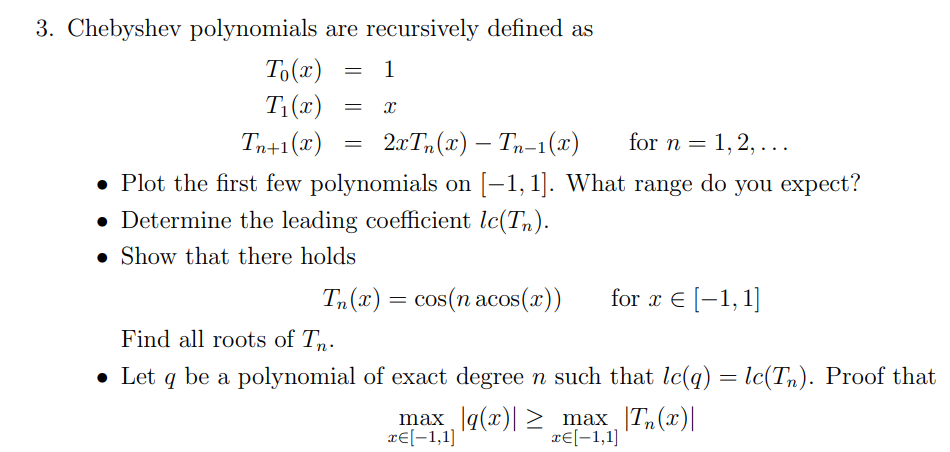
return max\_vals1, max\_vals2

plotting these for both types of points gives us:

**Conclusions:**

* The lagrange interpolation is a simple but numerically expensive ( O(n³) ) way to approximate functions
* Equally distributed points may have the tendency to oscillate around the original function. This tendency is known as Runge’s phenomenon and gets worse with an increasing number of points.
* One way of solving this issue is to use Chebyshev points

# Problem 3



**Plot:**

Chebyshev polynomials can be recursively defined in Python as follows:

def T\_n(x, n):

    if n == 0: return 1

    if n == 1: return x

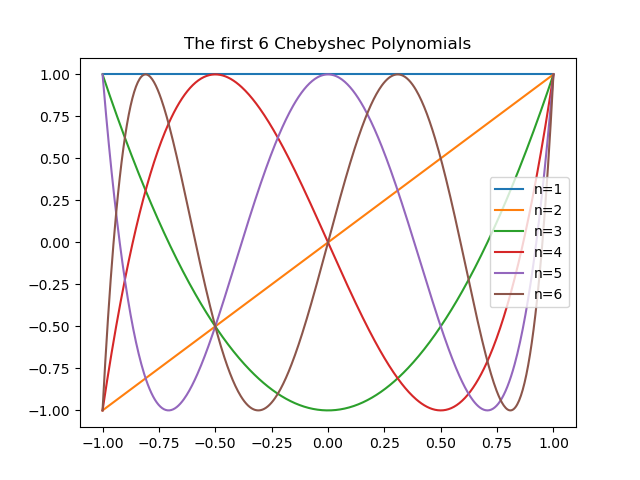
    return 2\*x\*T\_n(x,n-1) - T\_n(x, n-2)

Looking at : As long as every can only return a value between -1 and 1. Thus is multiplied by a number between -1 and 1. Therefore it can never be above 1. A maximum can be found at x = 1 and n = 2 =>

The minimum can be calculated the same by using x = -1 and n = 2 => min = -1

Thus we can expected polynomials to be in the range of [-1,1]. Beyond that will be multiplied by some number >1 which drives the function to infinity

Plotting the first polynomials for gives us:



**Leading Coefficients:**

To determine the leading coefficient, we can look at the recursive formula with. The expression results in the previous leading coefficient always to be doubled. This leads to the following pattern of leading coeffients: 1, 2, 4, 8, …

Therefore we can derive the following formula:

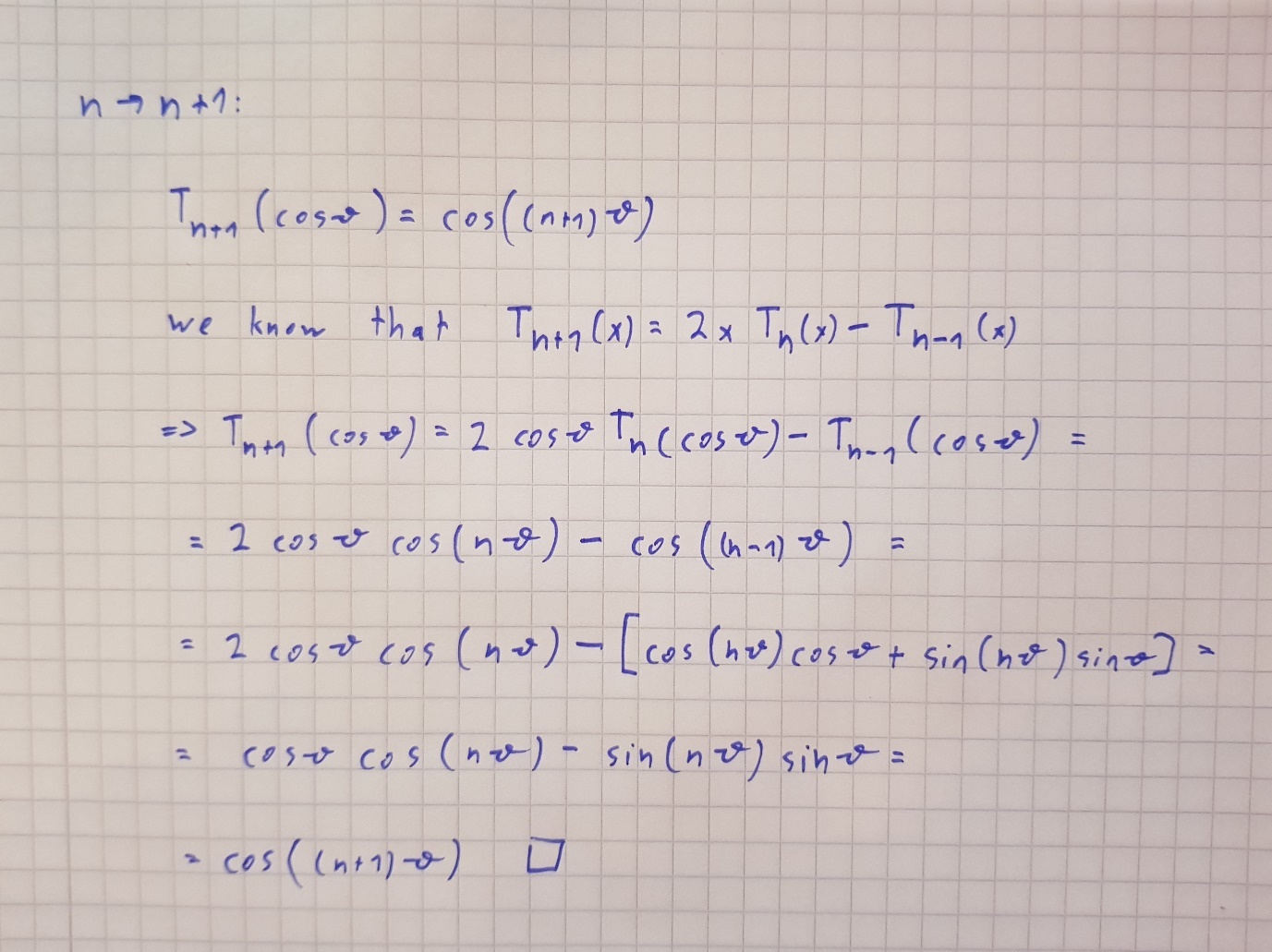
**Proof 1:**

We can show that



By using making a transformation from and using proof by induction

n=1:

**

**Roots:**

Using the transformation from before

The roots of the cosine function are given by:

**Proof2:**

Let Since q and Tn have the same leading coefficient p(x) is a polynomial of degree n-1

No we do proof by contradiction and assume: |q(x)| < |Tn (x)|

We can calculate the maxima of Tn with

We can do the same for he minima:

Therefore p(x) has to change sign at least n time on [-1, 1] which is not possible as p(x) is of degree n-1