Project 1 : Error bounds and linear system of equations

note: Functions are kept in an external SC_functions.py file. When referenced: "SC.function_name". I use some external functions i have made (called with the SC.function_name (for scientific computing)). What they do should be quite selv explanatory given their name. Its mostly just tidying up numpy operations.

In [73]:
```
# Imports
#Lets get the matrices and vectors
from watermatrices import Amat as A
from watermatrices import Bmat as B
from watermatrices import yvec as yvec
import numpy as np
import os
import importlib
import scipy

#Get some functions from external file
# Just direct to the file and import it as a module
cwd = os.getcwd()
os.chdir(r"C:\Users\sjefs\Desktop\Scientific Computing\Functions")
import SC_functions as SC
importlib.reload(SC)
os.chdir(cwd)
```

New things in resubmission version

Critique:

"Nice job Søren! Only comment is there were issues in your relative error for part 3 and 4 in G and it should have been a log plot"

What i have done:

- Question C part 1: I made a smarter, faster and simpler LU_factorization method.
- Question G : I fixed the relative error, and made it a log-plot

Week 1

Question a

Part 1

First we write a function for the infinity norm. This is just taking the sum of the absolute of each elements in a row, and figuring out which row has the highest sum. I do this with numpy sum function, where i can choose which axis to sum over.

Then i write the function calculating the condition number. Here i do the inverse with numpy linalg.

In [3]:
```
# We create a function for the infinity norm
def infinity_norm(M):
    return max(np.sum(abs(M), axis = 1))                    # We create a

#Test the functions (with external matrix generator)
print("testing infinity_norm")
M = SC.random_matrix(N = 5)
my_norm = infinity_norm(M)
real_norm = np.linalg.norm(M, np.inf)
print("My norm: ", my_norm)
print("numpy norm: ", real_norm)
if my_norm == real_norm: print("The function works")
print("")

# Now we create a function for the condition number
def condition_number(M):
    return infinity_norm(M) * infinity_norm(np.linalg.inv(M))

#Test the functions (with external matrix generator)
print("testing condition_number")
M = SC.random_matrix(N = 5)
my_cond = condition_number(M)
real_cond = np.linalg.cond(M, np.inf)
print("My norm: ", my_cond)
print("numpy norm: ", real_cond)
if my_cond == real_cond: print("The function works")
```

```
testing infinity_norm
My norm:  28.0
numpy norm:  28.0
The function works

testing condition_number
My norm:  40.673654121010024
numpy norm:  40.673654121010024
The function works
```

Part 2

Using the numpy block function, i stack the A and B matrixes to make E. Then we run our condition_number function for all of the $E - \omega S$ matrixes

We calculate the amount of significant digits with (found from exercise class TA):

$$cond \approx 10^{a-b} \implies b = a - log_{10}(cond)$$

with a and b being the number of input and output significant digits, and $cond$ being the condition number.

In [4]:
```
#Create the matrix E and S
E = np.block([[A, B], [B, A]])
S = np.zeros((14,14))
S[0:7, 0:7] = np.eye(7)
S[7:14, 7:14] = -np.eye(7)

#Define omega values
omega = np.array([0.8, 1.146, 1.400])

#Make the matrixes
test_matrixes = [E - x*S for x in omega]

#Calculate the bound
condition_numbers = [condition_number(matrix) for matrix in test_matrixes]
print("The condition numbers are: \n", *condition_numbers)
print("\n the amount of significant digits are : \n", 8 - np.log10(condition_num
```

```
The condition numbers are:
 327.81670424209915 152679.2687523386 227.19443667104446

 the amount of significant digits are :
 [5.48436892 2.81621993 5.64360231]
```

Here we round down the amount of significant digits.

Question b

Part 1

We already have the condition numbers, so we just need to calculate the fraction using our functions defined above.

In [5]:
```
delta_omega = 1/2*10**(-3)
result = [condition_numbers[i] * infinity_norm(delta_omega*S)/infinity_norm(E-om
print("The relative forward error bounds are: ", *result)
```

```
The relative forward error bounds are:  0.005220745069573262 2.4050352674535698
0.0035504027789308675
```

Question c

Part 1 !!!! NEW !!!!

To do LU factorization of a matrix $\mathbf{A}$, we first make a script for gaussian elemination. This revolves around finding a matrix $\mathbf{M_k}$ which eleminates all elements below the k_index of a vector. For a (4x4) matrix $\mathbf{A}$, continuing this, will give us an upper triangular matrix as: $\mathbf{M_3 M_2 M_1 A} = U$.

Now we can find $\mathbf{L} = AU^{-1}$, and have done our LU factorization

In [7]:
```
def el_mat(A, index):
    a = A[:, index]                 # Get the column
    n = np.shape(A)[0]
    M = np.eye(n)
    for i in range(index+1, n):     # Loop through the rows
```

```
        M[i, index] = -a[i]/a[index]
    return M

def gauss(A):
    A = np.copy(A)
    n = np.shape(A)[0]
    for i in range(n-1):
        M = el_mat(A, i)            # Get the elimination matrix
        A = np.dot(M, A)            # Apply the elemination matrix
    return A

def LU_factorization(A):
    # Now we just combine the two functions
    U = gauss(A)
    L = A @ np.linalg.inv(U)
    return L, U

# We test the function
A = SC.random_matrix(5)
A[1, 1] = 0
print("A \n", A)
L, U = LU_factorization(A)

test = np.round(L @ U, 4)

L = np.round(L, 4)
U = np.round(U, 4)
print("L \n", L)
print("U \n", U)
print("L*U = A \n", test)
```

```
A
 [[1. 8. 4. 0. 4.]
 [8. 0. 2. 9. 2.]
 [6. 4. 7. 0. 3.]
 [9. 1. 0. 2. 5.]
 [9. 5. 7. 3. 8.]]
L
 [[ 1.      0.      0.      0.      0.    ]
 [ 8.      1.      0.     -0.      0.    ]
 [ 6.      0.6875  1.     -0.      0.    ]
 [ 9.      1.1094 -0.75    1.      0.    ]
 [ 9.      1.0469  0.6638  0.1833  1.    ]]
U
 [[  1.       8.       4.       0.       4.    ]
 [  0.     -64.     -30.       9.     -30.    ]
 [  0.       0.       3.625   -6.1875  -0.375 ]
 [  0.       0.       0.     -12.625    2.    ]
 [  0.       0.       0.       0.       3.2885]]
L*U = A
 [[1. 8. 4. 0. 4.]
 [8. 0. 2. 9. 2.]
 [6. 4. 7. 0. 3.]
 [9. 1. 0. 2. 5.]
 [9. 5. 7. 3. 8.]]
```

Part 2 and part 3

We create the forward substitution function according to:

$$x_1 = \frac{b_1}{\ell_{11}}, \quad x_i = \left(b_i - \sum_{j=1}^{i-1} \ell_{ij} x_j\right) / \ell_{ii}, \quad i = 2, \ldots, n.$$

This means with a matrix U and a vector $\vec{b}$ of dimension n, we start by finding the first x_1 value. Then we take a step forward to x_2 which can now be found using x_1. We continue this scheme until we have all the elements of vector $\vec{x}$

And the backward substitution function according to:

$$x_n = \frac{b_n}{u_{nn}}, \quad x_i = \left(b_i - \sum_{j=i+1}^{n} u_{ij} x_j\right) / u_{ii}, \quad i = n-1, \ldots, 1.$$

This means with a matrix U and a vector $\vec{b}$ of dimension n, we start by finding the last x_n value. Then we take a step back to x_{n-1} which can now be found using x_n. We continue this scheme until we have all the elements of vector $\vec{x}$

In [8]:
```
# We create the forward substitution function
def forward_substitute(L, b):
    n = len(b)-1
    x = np.zeros(len(b))
    x[0] = b[0]/L[0, 0]

    for i in range(1, n + 1):
        x[i] = (1/L[i, i])*(b[i] -  sum([ L[i, j]*x[j] for j in range(0, n)])
    return x

# We create the backwards substitution function
def back_substitute(U, b):
    n = len(b)-1
    x = np.zeros(len(b))
    x[n] = b[n]/U[n, n]

    for i in range(n-1, -1, -1):
        x[i] = (1/U[i, i])*(b[i]- sum([U[i, j]*x[j]  for j in range(1, n+1)])
    return x


## And we test it on the linear equation
M = np.array([[2, 1, 1], [6, 1, 4], [-6, -5, 3]])
b = [4, 11, 4]

#First we factorize
L, U = SC.LU_factorization(M)

#Then we use forward substitution to get y
y = SC.forward_substitute(L, b)
# and backwards substitution to get x
x = SC.back_substitute(U, y)
print("My solution is: \n", x)
real = np.linalg.solve(M, b)
print("The real solution is: \n", real)
if np.allclose(x, real): print("\n The function works")
```

```
My solution is:
 [-0.8  2.2  3.4]
The real solution is:
 [-0.8  2.2  3.4]

 The function works
```

Problem d

Part 1

First we make a function combining all of our previous functions to solve linear equations

The we use that one in creating the solve_alpha function

In [9]:
```
# We start by creating the z vector:
z = np.block([yvec, -yvec])

# Then we combine our previous functions:
def LU_solver(M, b):
    L, U = LU_factorization(M)
    y = forward_substitute(L, b)
    x = back_substitute(U, y)
    return x

# And create a function for calculating alpha given an omega value
def solve_alpha(E, S, z, w):
    new_matrix = E - w*S
    x = LU_solver(new_matrix, z)
    return np.dot(z, x)

# We test it against numpy library
omega_test = 0.8
my_result = SC.solve_alpha(E, S, z, omega_test)
real = np.dot(np.linalg.solve(E - omega_test*S, z), z)
print("testing solve_alpha:")
if np.allclose(my_result, real): print("The function works")
```

```
testing solve_alpha:
The function works
```

I am not completely sure what the problem asks me to do. My interpretation is, that we look at the ω values themselves, and then at the boundaries $\pm\delta\omega$

In [10]:
```
import matplotlib.pyplot as plt
omega = np.array([0.8, 1.146, 1.400])
delta_omega = 1/2*10**(-3)
omega_minus = omega - delta_omega
omega_plus = omega + delta_omega

alpha_single = [SC.solve_alpha(E, S, z, w) for w in omega]
alpha_minus = [SC.solve_alpha(E, S, z, w) for w in omega_minus]
alpha_plus = [SC.solve_alpha(E, S, z, w) for w in omega_plus]

table = np.array([alpha_single, alpha_minus, alpha_plus]).T
table = np.round(table, 4)
omega_labels = [r"$ \alpha (\omega)$", r"$\alpha (\omega - \delta \omega) $", r"
alpha_labels = [r"$\omega = 0.8$", r"$\omega = 1.146$", r"$\omega = 1.4$"]
```

```
fig, ax = plt.subplots(figsize=(8, 2))
ax.axis('off')
table = ax.table(cellText=table, colLabels=omega_labels, rowLabels=alpha_labels,
table.auto_set_font_size(False)
table.set_fontsize(12)
table.scale(1, 2)
ax.set_title(r'$\omega$')
plt.show()
```

ω

	$\alpha(\omega)$	$\alpha(\omega - \delta\omega)$	$\alpha(\omega + \delta\omega)$
$\omega = 0.8$	1.6361	1.6278	1.6444
$\omega = 1.146$	2609.2353	994.753	-4185.0183
$\omega = 1.4$	-2.7069	-2.7139	-2.6999

Part 2

I would use (b). We want to be able to examine the behavior of the polarizability subject to small pertubations. (b) is the only one that accounts for the pertubation $\delta\omega$. Also, calculating $\alpha(\omega)$ and evaluating bound on alpha, is essentially evaluating a forward error

Part 3

So lets move things a bit around. Let start by looking at $\alpha(\omega) = z \cdot x$. Looking at z we note that $z \cdot z = |z|^2$, which means that: $\frac{z \cdot z}{|z|^2} = 1$, and we can reconize an inverse $(z)^{-1} = \frac{z}{|z|^2}$, so that $x = z^{-1}\alpha(\omega)$

Now we can use (we denote all infinity norms as $||x||$):

$$||\Delta x|| = ||\Delta(\alpha z^{-1})|| = ||z^{-1}||\ \ ||\Delta\alpha(\omega)||$$

Because the z vector is just a constant. In a similair fasion:

$$||x|| = ||\alpha z^{-1}|| = ||z^{-1}||\ \ ||\alpha(\omega)||$$

We also note that $||S||$ is just 1 because S consists of identity matrices. This means that $||\delta\omega S|| = |\delta\omega|$

And we get our final result, by substituting and moving around with the bound equation in question b:

$$|\Delta\alpha(\omega)| <= cond(E - \omega S)\frac{|\alpha(\omega)|}{||E-\omega S||}|\delta\omega|$$

In [11]:
```
# Lets calculate a condition number for each omega value
def alpha_cond(w):
    return abs(SC.condition_number(E - w*S) * (SC.solve_alpha(E, S, z, w))/(SC.i
alpha_cond_single = [alpha_cond(w) for w in omega]
print("Our error bounds are", alpha_cond_single)
```

```
#And check it with our values
relative_errors_backwards = [abs(alpha_plus[i] - alpha_single[i]) for i in range
relative_errors_forwards = [abs(alpha_single[i] - alpha_minus[i]) for i in range
print("\n the calculated relative errors are:")
print(relative_errors_backwards)
print(relative_errors_forwards)
```

```
Our error bounds are [np.float64(0.008541860088297487), np.float64(6275.302974922
086), np.float64(0.009610559215234522)]

 the calculated relative errors are:
[np.float64(0.008293070202700425), np.float64(6794.253607222388), np.float64(0.00
6965852003208095)]
[np.float64(0.0083224336057095), np.float64(1614.4823322103439), np.float64(0.006
987455515579555)]
```

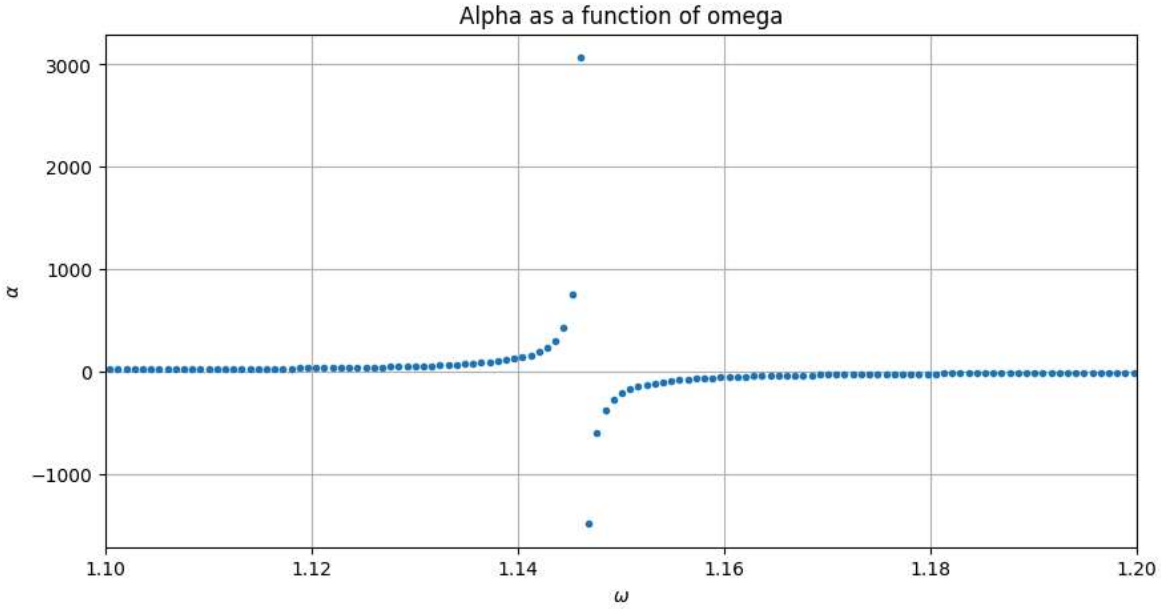
We note that around the resonance frequency, pertubating with $+\delta\omega$ actually exceeds our bound. I think this problem arises as our matrix A becomes near singular around this singularity, which makes it difficult to evaluate an inverse matrix.

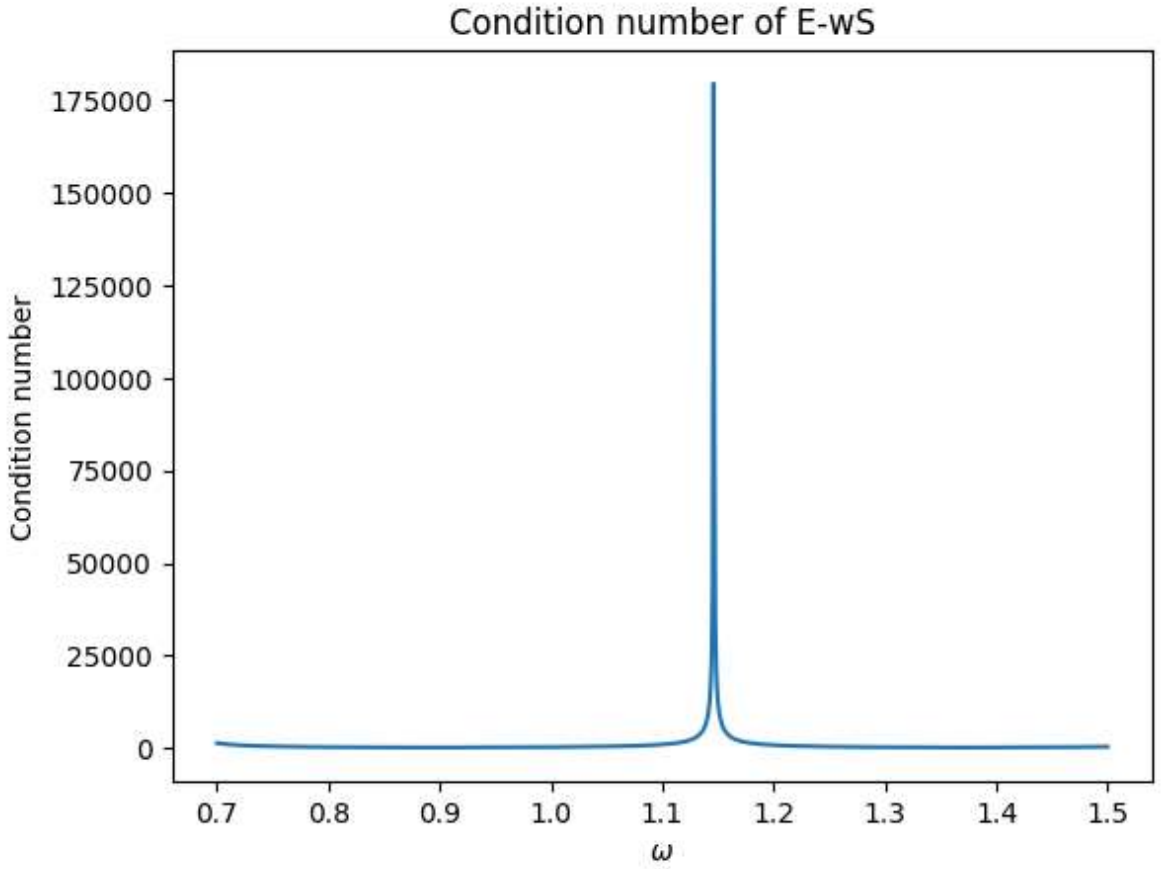
Question e

part 1

In [12]:
```
omega_test = np.linspace(0.7, 1.5, 1000)

alpha_test = np.array([SC.solve_alpha(E, S, z, w) for w in omega_test])
plt.figure(figsize = (10, 5))
plt.title("Alpha as a function of omega")
plt.plot(omega_test, alpha_test, '.')
plt.xlim(1.1, 1.2)
plt.xlabel(r'$\omega$')
plt.ylabel(r'$\alpha$')
plt.grid()

plt.figure()
plt.title("Condition number of E-wS")
plt.xlabel(r'$\omega$')
plt.ylabel("Condition number")
toplot = [SC.condition_number(E-w*S) for w in omega_test]
plt.plot(omega_test, toplot)
```

Out[12]:
```
[<matplotlib.lines.Line2D at 0x2d8c97a0050>]
```





Part 2

We note that, around ω = 1.146 the condition number of the matrix $\mathbf{E} - \omega\mathbf{S}$ becomes very large, thus small pertubations around this ω value, will result in a large forward error on α. This could also mean that the matrix of the linear system of equations approaches almost singularity.

Week 2

Question f

Part 1

We create this algorithm to work as in example 3.8. It works for the eqations in example 3.1, and for most randomly generated matrixes (those without specific annoying zeros)

In [31]:
```
def householder_QR_slow(A):
    M = np.copy(A)
    H_storage = []

    for i in range(np.shape(M)[1]):                                    #Loop
        a = np.copy(M.T[i])
        a[:i] = 0                                                      #We c
        alpha = -np.sign(a[i])*np.linalg.norm(a)                       #Calc

        v = a - alpha * np.eye(np.shape(M)[0])[i]                      #Crea
        H = np.eye(np.shape(M)[0]) - 2 * np.outer(v, v) / np.dot(v, v) #Crea

        H_storage.append(H)                                            #Stor
        M = np.matmul(H, M)                                            #upda

    R = np.triu(M)                                                     #Extr
    Q = np.transpose(np.linalg.multi_dot(H_storage))                   #Calc
    return Q, R

#Example from book to test it on.
A, b  = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1], [-1, 1, 0], [-1, 0, 1], [0, -
#A = np.random.randint(-10, 10, size = (6, 3))   #Or try a random matrix (with s
print(A)
Q, R = householder_QR_slow(A)
if np.allclose(np.matmul(np.transpose(Q), Q), np.eye(6)): print("Q is a orthogon
if np.allclose(np.round(np.matmul(np.transpose(Q), R), 8), A): print("QR = A and
```

```
[[ 1  0  0]
 [ 0  1  0]
 [ 0  0  1]
 [-1  1  0]
 [-1  0  1]
 [ 0 -1  1]]
Q is a orthogonal matrix of dimensions (6, 6)
QR = A and R is a matrix of dimensions (6, 3)
```

Part 2

To combine the vectors v and the upper triangular matrix R, we add a zero-row in the beginning of the V matrix, and add a zero-row at the and of R. Then simply adding them together should give the (m+1, n) matrix VR. (We use 'V' for the matrix consisting of the vectors 'v'). We also create a small function for extracting the VR matrix into V and R seperately

In [33]:
```
def householder_fast(A):
    M = np.copy(A)
    V = np.array([]).reshape(np.shape(M)[0], 0)                        #Crea

    for i in range(np.shape(M)[1]):                                    #Loop
```

```
        a = np.copy(M.T[i])
        a[:i] = 0                                                    #We c
        alpha = -np.sign(a[i])*np.linalg.norm(a)                     #Calc

        v = a - alpha * np.eye(np.shape(M)[0])[i]                    #Crea
        V = SC.add_column(V, column = v)                             #Stor
        H = np.eye(np.shape(M)[0]) - 2 * np.outer(v, v) / np.dot(v, v)     #Crea

        M = np.matmul(H, M)                                          #upda

    R = np.triu(M)                                                   #Extr
    R = np.vstack([R, np.zeros(np.shape(R)[1])])                     #Add
    V = SC.add_row(V, index = 0)                                     #Add
    VR = R + V                                                       #Add
    return VR

def VR_extract(VR):
    R = np.triu(VR)                                          #Extract the upper
    V = (VR-np.triu(VR))[1:]
    R = R[:np.shape(R)[1]]
    return R, V
VR = householder_fast(A)
print(VR, "It works !")
R, V = VR_extract(VR)
```

```
[[-1.73205081  0.57735027  0.57735027]
 [ 2.73205081 -1.63299316  0.81649658]
 [ 0.          2.63299316 -1.41421356]
 [ 0.          0.          2.41421356]
 [-1.          0.78867513  0.03324491]
 [-1.         -0.21132487  0.72314286]
 [ 0.         -1.          0.68989795]] It works !
```

Part 3

This is done as in the last step of example 3.8. instead of using the H matrices after each other, we use the function p. 123 to calculate $\mathbf{Ha}$ as a function of v and a.

In [35]:
```
VR = householder_fast(A)
R, V = VR_extract(VR)

def least_squares(A, b):
    a = np.copy(b)
    VR = householder_fast(A)                                 # We perform the
    R, V = VR_extract(VR)                                    # We extract the

    for v in V.T:                                            # We use the hous
        a = a - np.matmul(2*np.outer(v, a)/np.dot(v,v), v)

    n = np.shape(R)[1]
    x = SC.back_substitute(R, a[:n])                     # We solve the system
    r = a[n:]
    return x, r


x, r = least_squares(A, b)
print(x, r, "This is consistent with the answer from example 3.8 !! \n")

# Lets test it against the example !
```

```
from HHexamples import A1, b1, x1
x, r = least_squares(A1, b1)
R = VR_extract(householder_fast(A1))[0]
print("the upper triangular matrix is: \n", R)
print("The solution is: ",  np.round(x, 3))
print("with a residual of: ", r)
```

```
[1236. 1943. 2416.] [ 4.95132607 -3.09403081  0.95464312] This is consistent with
the answer from example 3.8 !!

the upper triangular matrix is:
 [[-5.91607978 -7.43735744]
 [ 0.          0.82807867]]
The solution is:  [0.  0.5]
with a residual of:  [-2.22044605e-16]
```

Question g

Part 1

I suggest $\omega_p = 1.144$ to avoid the instability around the resonance frequency around $1.146307999

Part 2 and 3

To prepare for the least squares fit, we want to combine many equations on the form: $\alpha(\omega_i) = [1, \omega_i^2, \omega_i^4, \omega_i^6, \omega_i^8]^T \cdot [a_0, a_1, a_2, a_3, a_4]$ Here $\alpha(\omega_i)$ refers to the value of α calculated using our function at the specific datapoint ω_i.

To combine these, we just arrange a vector $\alpha = [\alpha(\omega_0), \alpha(\omega_1), \ldots, \alpha(\omega_n)]$, a vector of the fit parameters $\mathbf{a} = [a_0, a_1, a_2, a_3, a_4]$, and a matrix $\mathbf{A}$ consisting of stacked rows of $[1, \omega_i^2, \omega_i^4, \omega_i^6, \omega_i^8]$, so the equation takes the form:

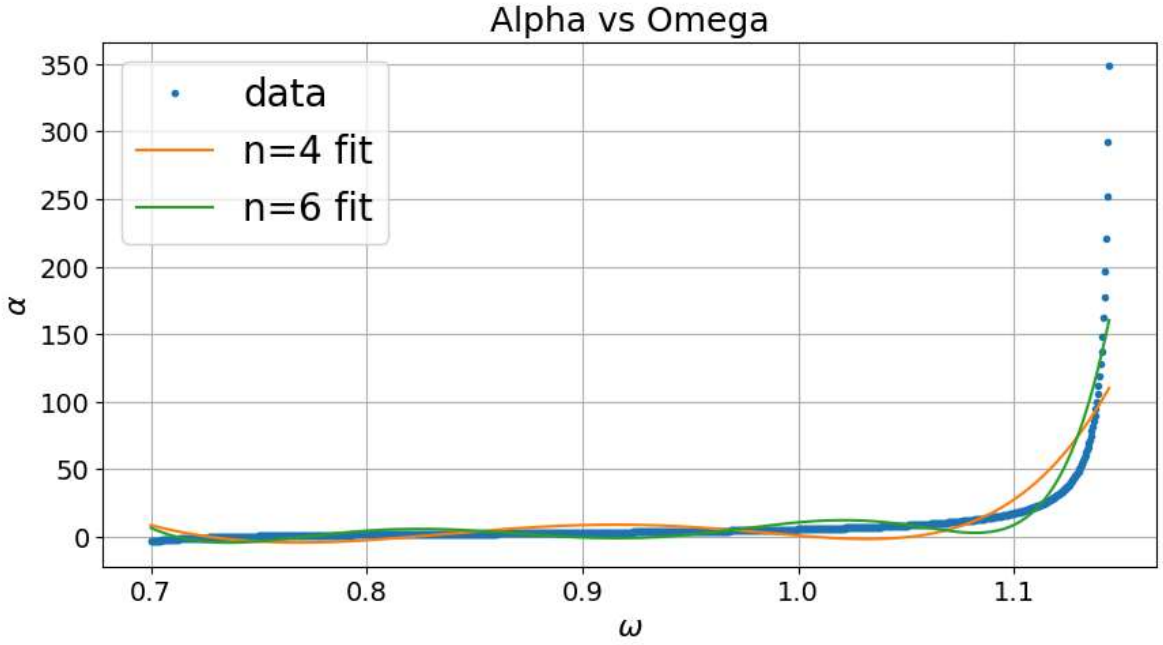
$$\mathbf{A}\mathbf{a} = \alpha(\omega)$$

We plot the function lastly

In [97]:
```
# We create our data
w_p = 1.144

omega_vector = np.linspace(0.7, w_p, 1000)
alpha_vector = np.array([SC.solve_alpha(E, S, z, w) for w in omega_vector])

# Solve n = 4
A_matrix_4 = np.array([[w**(2*i) for i in range(5)] for w in omega_vector])
a_4, r_4 = SC.least_squares(A_matrix_4, alpha_vector)
alpha_4 = np.matmul(A_matrix_4, a_4)
print("The solution for n = 4 is: \n", a_4)

# Solve n = 6
A_matrix_6 = np.array([[w**(2*i) for i in range(7)] for w in omega_vector])
a_6, r_6 = SC.least_squares(A_matrix_6, alpha_vector)
alpha_6 = np.matmul(A_matrix_6, a_6)
print("The solution for n = 6 is: \n", a_6)

#plot the solutions
```

```
plt.figure(figsize = (10, 5))
plt.plot(omega_vector, alpha_vector, '.', label = "data")
plt.plot(omega_vector, alpha_4, label = "n=4 fit")
plt.plot(omega_vector, alpha_6, label = "n=6 fit")

plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.xlabel(r'$\omega$', fontsize=16)
plt.ylabel(r'$\alpha$', fontsize=16)
plt.title('Alpha vs Omega', fontsize=18)
plt.legend(fontsize=20)
plt.grid()
plt.show()
```

```
The solution for n = 4 is:
 [  1492.18826806  -7855.10756201  15030.68019872 -12396.48143307
   3729.64943165]
The solution for n = 6 is:
 [  19730.42921319 -152900.96417146  483532.36801701 -798971.32099273
  727959.18404893 -347017.37617015   67678.41545665]
```



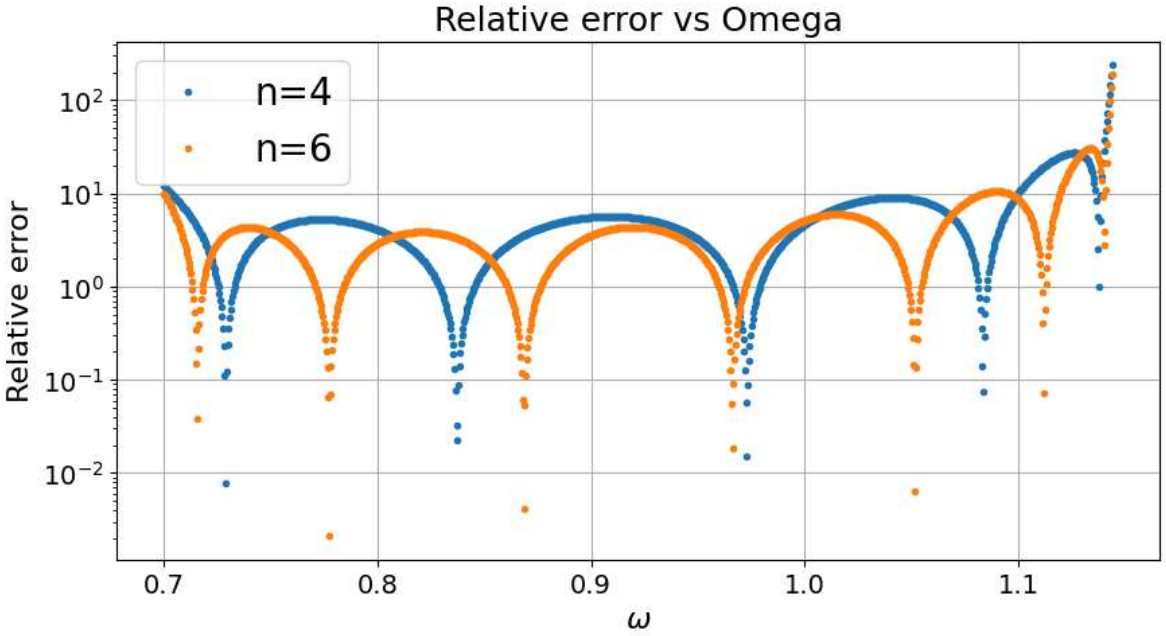
Part 3 !!! New !!!

We calculate the relative errors and plot them side by side

In [101…

```
# Lets compare relative errors


relative_error_4 = abs(alpha_4 - alpha_vector)   #Perhaps use norm


relative_error_6 = abs(alpha_6 - alpha_vector)


plt.figure(figsize = (10, 5))
plt.plot(omega_vector, relative_error_4, '.', label = "n=4")
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.plot(omega_vector, relative_error_6, '.', label = "n=6")
plt.yscale('log')
```

```
plt.xlabel(r'$\omega$', fontsize=16)
plt.ylabel('Relative error', fontsize=16)
plt.title('Relative error vs Omega', fontsize=18)
plt.legend(fontsize=20)
plt.grid()
plt.show()
```



Part 4

We use the psoudeinverse to calculate the condition number of the A matrix.

In [102…

```
def condition_number_2(A):
    return np.linalg.norm(A, ord = np.inf)*np.linalg.norm(np.linalg.pinv(A), ord

omega_vector = np.linspace(0.7, w_p, 1000)
alpha_vector = np.array([SC.solve_alpha(E, S, z, w) for w in omega_vector])

cond_4 = condition_number_2(A_matrix_4)
cond_6 = condition_number_2(A_matrix_6)
print("The condition number for n = 4 is: ", cond_4)
print("The condition number for n = 6 is: ", cond_6)

def digits(cond):
    return 8-np.log10(cond)

print("The amount of significant digits for n = 4 is: ", digits(cond_4))
print("The amount of significant digits for n = 6 is: ", digits(cond_6))
```

```
The condition number for n = 4 is:  17225.510638960826
The condition number for n = 6 is:  2328691.520773773
The amount of significant digits for n = 4 is:  3.7638278948545416
The amount of significant digits for n = 6 is:  1.632888038171556
```

Question h

Part 1

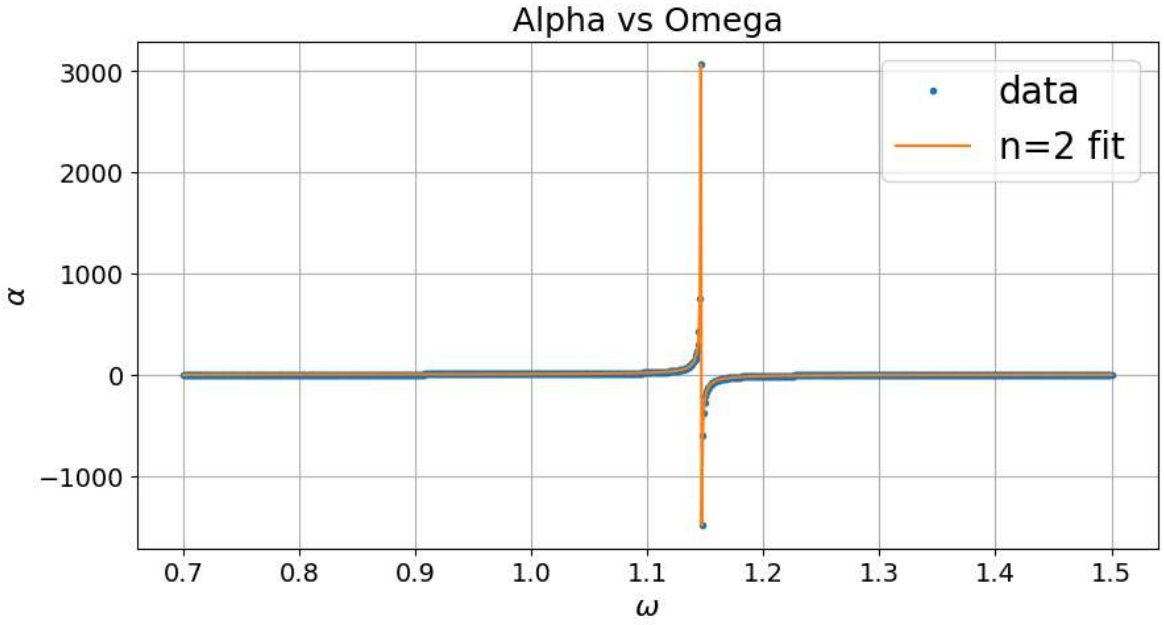
equation 5 is not able to handle singularities, as it is just a polynomial of even exponentials, which cant be symetrical about a value of ω. The new $Q(\omega)$ deals with uneven terms of polynomials.

Part 2

WE use the function from the hint for lineralizing $Q(\omega)$.

In [39]:
```
def Q(w, n = 2):
    list_1 = [w**i for i in range(n+1)]
    list_2 = [-solve_alpha(E, S, z, w)*w**i for i in range(1, n+1)]
    return np.append(list_1, list_2)

omega_vector = np.linspace(0.7, 1.5, 1000)
alpha_vector = np.array([SC.solve_alpha(E, S, z, w) for w in omega_vector])

A_matrix_Q_2 = [Q(w, n = 2) for w in omega_vector]
b = np.array([solve_alpha(E, S, z, w) for w in omega_vector])

a_q_2, r_q_2 = SC.least_squares(A_matrix_Q_2, b)
print(a_q_2)
plt.figure(figsize = (10, 5))
plt.plot(omega_vector, alpha_vector, '.', label = "data")
plt.plot(omega_vector, np.matmul(A_matrix_Q_2, a_q_2), label = "n=2 fit")
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.grid()
plt.xlabel(r'$\omega$', fontsize=16)
plt.ylabel(r'$\alpha$', fontsize=16)
plt.title('Alpha vs Omega', fontsize=18)
plt.legend(fontsize=20)
plt.show()
```

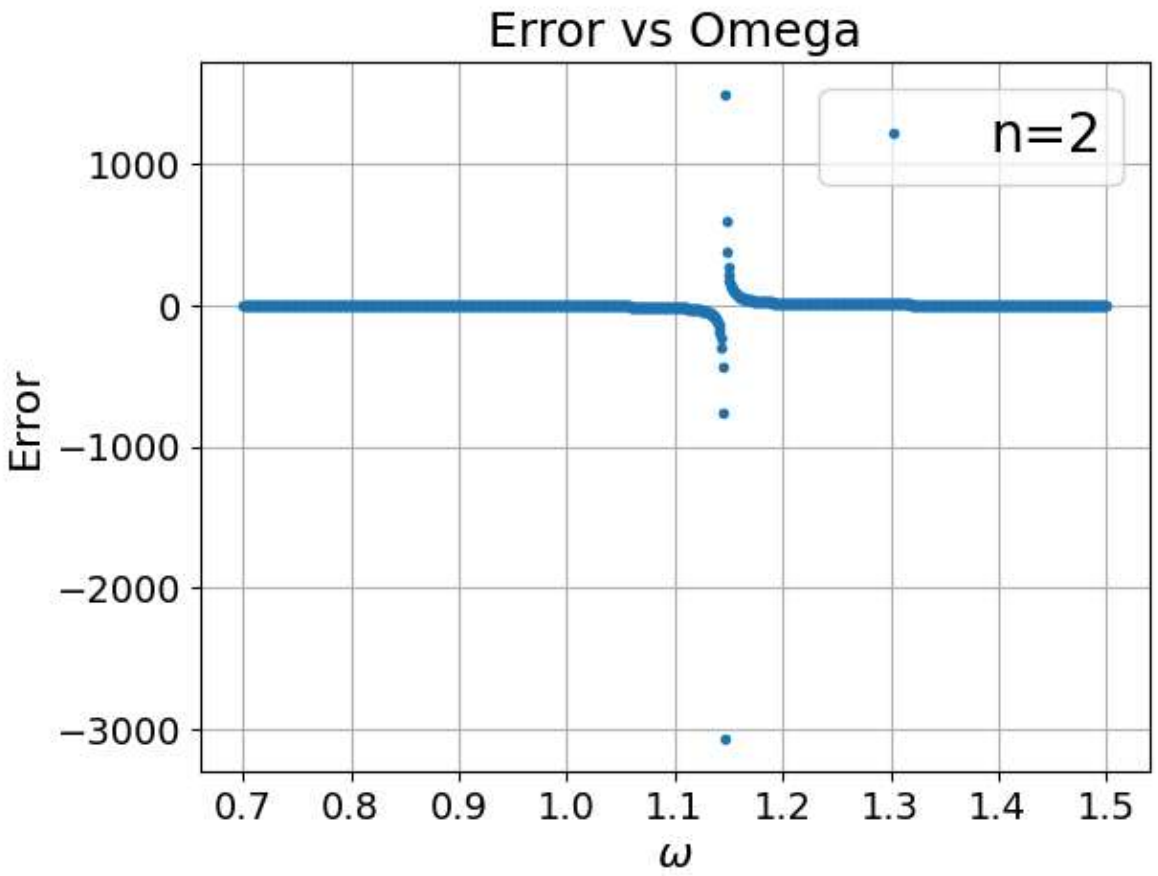
```
[ 1.92844516 -3.45255251  1.1443205  -2.39367192  1.32713323]
```



Lets do the errors :

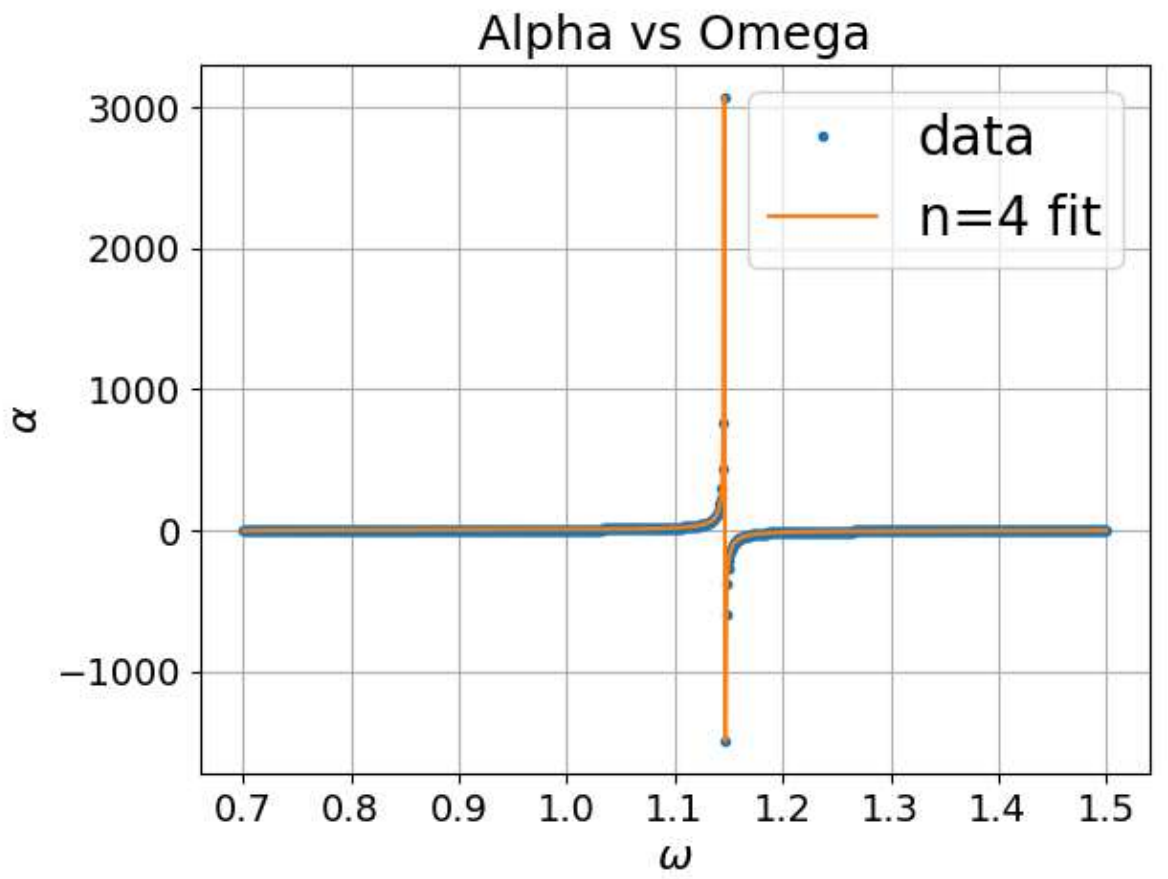
In [40]:
```
def Q_real(parameters, w, n):
    index = n + 1
```

```
    a_coefficients = parameters[:index]
    b_coefficients = parameters[index:]
    upper = sum([a_coefficients[i] * w**i for i in range(n+1)])
    lower = 1 + sum([b_coefficients[i] * w**i for i in range(1, n)])
    return upper/lower

error_Q_2 = Q_real(a_q_2, omega_vector, 2) - alpha_vector
plt.plot(omega_vector, error_Q_2, '.', label = "n=2")
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.grid()
plt.xlabel(r'$\omega$', fontsize=16)
plt.ylabel('Error', fontsize=16)
plt.title('Error vs Omega', fontsize=18)
plt.legend(fontsize=20)
plt.show()
```

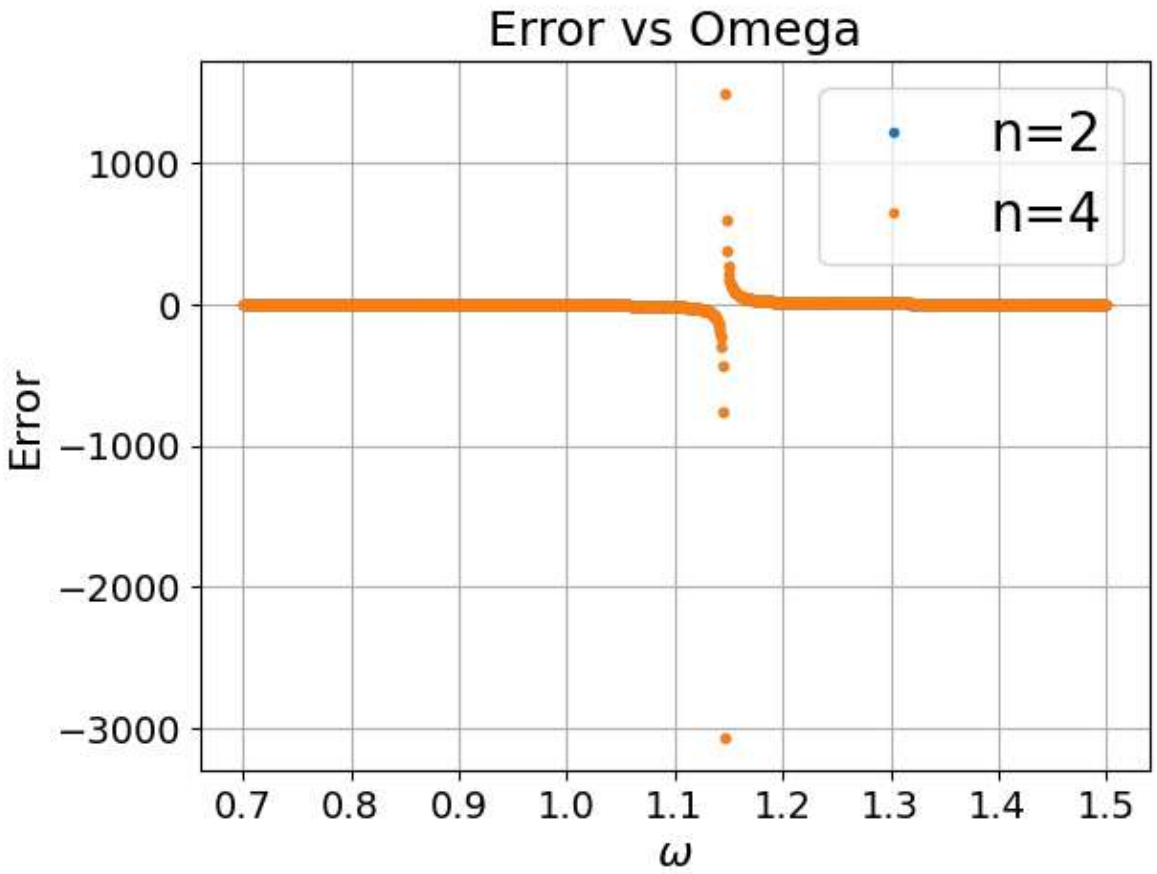


In [41]:
```
omega_vector = np.linspace(0.7, 1.5, 1000)
alpha_vector = np.array([SC.solve_alpha(E, S, z, w) for w in omega_vector])

A_matrix_Q_4 = [Q(w, n = 4) for w in omega_vector]
b = np.array([solve_alpha(E, S, z, w) for w in omega_vector])

a_q_4, r_q_4 = SC.least_squares(A_matrix_Q_4, b)


plt.plot(omega_vector, alpha_vector, '.', label = "data")
plt.plot(omega_vector, np.matmul(A_matrix_Q_4, a_q_4), label = "n=4 fit")
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.grid()
plt.xlabel(r'$\omega$', fontsize=16)
plt.ylabel(r'$\alpha$', fontsize=16)
```

```
plt.title('Alpha vs Omega', fontsize=18)
plt.legend(fontsize=20)
plt.show()
```



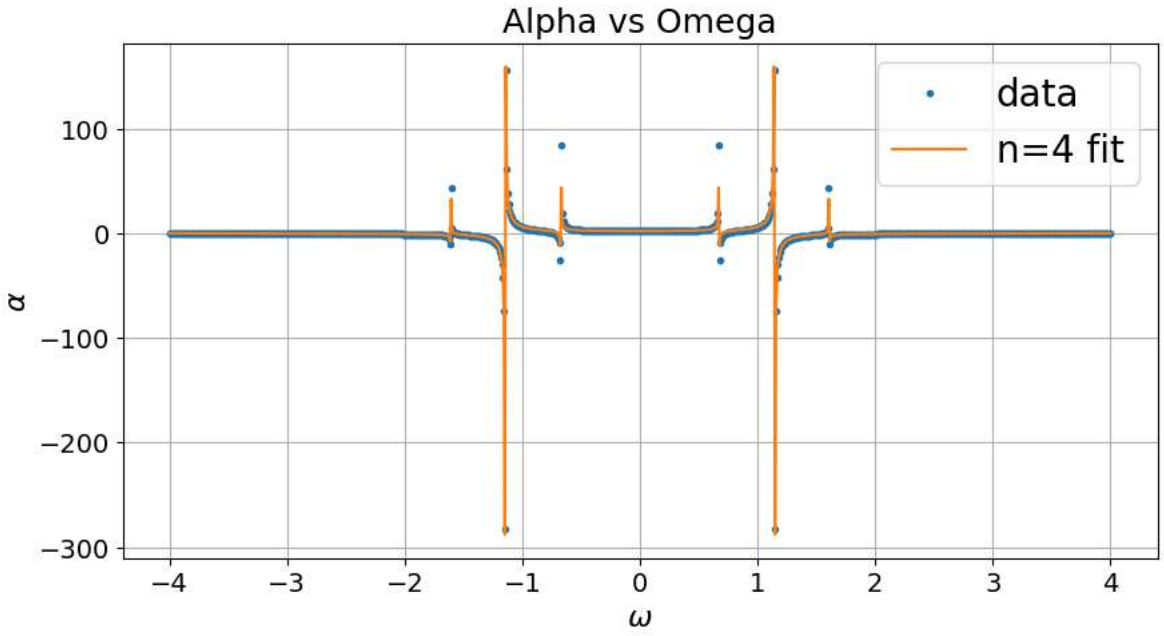
In [42]:
```
error_Q_4 = Q_real(a_q_4, omega_vector, 4) - alpha_vector
plt.plot(omega_vector, error_Q_2, '.', label = "n=2")
plt.plot(omega_vector, error_Q_4, '.', label = "n=4")
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.grid()
plt.xlabel(r'$\omega$', fontsize=16)
plt.ylabel('Error', fontsize=16)
plt.title('Error vs Omega', fontsize=18)
plt.legend(fontsize=20)
plt.show()
```



Part 3

WE extend to the whole range. It seems like the function is able to catch some of this behavior. My idea is, that we want denominator in Q to be zero around a singularity, and switch sign after a singularity. A function that should be able to catch singularities, would prabably have some terms like $(\omega - k_0)(\omega - k_1)(\omega - k_2)$ in the denominator, so we can fit for the k_i values, which should be located around the frequency of the singularities.

In [43]:
```
omega_vector = np.linspace(-4, 4, 1000)
alpha_vector = np.array([SC.solve_alpha(E, S, z, w) for w in omega_vector])

A_matrix_Q_4 = [Q(w, n = 4) for w in omega_vector]
b = np.array([solve_alpha(E, S, z, w) for w in omega_vector])

a_q_4, r_q_4 = SC.least_squares(A_matrix_Q_4, b)

plt.figure(figsize = (10, 5))
plt.plot(omega_vector, alpha_vector, '.', label = "data")
plt.plot(omega_vector, np.matmul(A_matrix_Q_4, a_q_4), label = "n=4 fit")
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.grid()
plt.xlabel(r'$\omega$', fontsize=16)
plt.ylabel(r'$\alpha$', fontsize=16)
plt.title('Alpha vs Omega', fontsize=18)
plt.legend(fontsize=20)
plt.show()
```


Alpha vs Omega
data
n=4 fit
100
0
−100
−200
−300
α
−4
−3
−2
−1
0
1
2
3
4
ω