Scientific Computing Project 1

Jakob Schauser, pwn274

September 2023

a

(1)

Implementing the condition number was easy given their definitions as we have been taught:

```
maxnorm = lambda M: np.abs(M).sum(axis=1).max()

condition_number = lambda M: maxnorm(M) * maxnorm(np.linalg.inv(M))
```

(2)

For $\omega = 0.800, 1.146, 1.400$ i get $327.81670424209915, 152679.2687523386, 227.19443667104446$ respectively

Using definition on page 60, I can find maximal number of significant digits (assuming that we are in base ten) given that everything else is perfect to be $\lfloor \log_{10}() \rfloor$.

```
significant_digits = int(np.log10(condition_number(E-w*S)))
```

which gives (5,2,5) respectively

b

1

Inserting our system of equations into the given bound, we can find:

$$\frac{\|\Delta x\|_\infty}{\|x\|_\infty} = \text{cond}_\infty(\mathbf{A}) \frac{\|\delta \mathbf{A}\|_\infty}{\|\mathbf{A}\|_\infty} = \text{cond}_\infty(\mathbf{E} - \omega \mathbf{S}) \frac{\|\delta \omega \mathbf{S}\|_\infty}{\|\mathbf{E} - \omega \mathbf{S}\|_\infty} \tag{1}$$

Finding the significant figures, I combine this with the equation from b.(2):

$$\text{significant digits} = -\log_{10}\left(\frac{\|\Delta x\|_\infty}{\|x\|_\infty}\right) = -\log_{10}\left(\text{cond}_\infty(\mathbf{E} - \omega \mathbf{S}) \frac{\|\delta \omega \mathbf{S}\|_\infty}{\|\mathbf{E} - \omega \mathbf{S}\|_\infty}\right) = \tag{2}$$

This gives me the following table where $b \equiv \text{cond}_\infty(\mathbf{E} - \omega \mathbf{S}) \frac{\|\delta \omega \mathbf{S}\|_\infty}{\|\mathbf{E} - \omega \mathbf{S}\|_\infty}$:

ω	b	$-\log_{10}(b)$	significant digits
0.8	0.01044	1.884328	1
1.146	4.81	-0.622967	-1
1.4	0.007101	2.294820	2

I have also added the second method of estimating the number of significant digits, but I am unsure about the validity of using a bound in this way.

c

I have implemented the following algorithms:

(1) lu_factorize

Basing my code upon the pseudocode in the book, choosing (in order to better understand it myself) to also implement partial pivoting, I get the following:

```
def decomp(M, should_pivot=True):
    # There ws some code here that i have removed for clarity

    for i in range(M.shape[0]-1):
        # pivot
        P = I()
        if should_pivot:
        # find best row
            best = np.argmax(U[i:,i]) + i # compensate for looking at submatrix

            # swap
            P[[i,best]] = P[[best,i]]

            # apply
            U = P@U

        # save for generating L
        Ps[i] = P

        # eliminate
        coeff = I()

        # find coefficients
        cc = -U[(i+1):,i]/U[i,i]
        coeff[(i+1):,i] = cc

        # apply
        U = coeff@U

        # save for generating L
        coeff[(i+1):,i] = -cc
        Ls[i] = coeff

    # generate L
    L = I()
    for ii in range(len(Ps))[::-1]:
        L = Ls[ii]@L
        L = Ps[ii].T@L


    assert same(L@U, M), "Decomposition did not work"

    return U, L
```

Sorry if this is too much code, I have a hard time cutting down and still maintaining clarity

(2) forward_substitute & (3) back_substitute

Simply implemented as follows:

```
def forward_substitute(L, z):
```

```
    y = np.empty_like(z)

    for k in range(L.shape[0]):
        y[k] = z[k] - np.dot(L[k, 0:k], y[0:k])

    return y

def backward_substitute(U, y):
    x = np.empty_like(y)
    N = U.shape[0]

    for k in range(N)[::-1]:
        x[k] = (y[k] - np.dot(U[k, (k+1):N], x[(k+1):N])) / U[k,k]

    return x
```

I tested The array of created function them all on the following system of equations:

$$\begin{bmatrix} 2 & 1 & 1 \\ 4 & 1 & 4 \\ -6 & -5 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 11 \\ 4 \end{bmatrix} \tag{3}$$

giving me:

$$\mathbf{v} = \begin{bmatrix} -4 \\ 7 \\ 5 \end{bmatrix} \tag{4}$$

on which Numpy also agrees.

d

1 & 2 & 3

Solving for alpha

```
# this is used all throughout my code
def same(a,b):
    return np.isclose(a,b).all()

# Generic linear algebra solver using my previously defined functions
def solve(M, z):
    U, L = lu_factorize(M, should_pivot=False)
    y = forward_substitute(L, z)
    x = backward_substitute(U, y)
    assert same(x, np.linalg.solve(M,z)), "Numpy says solution is wrong"
    assert same(M@x, z), "x is not a solution"
    return x

# Find the alpha given an omega
def solve_alpha(omega):
    M = data["E"]-omega*data["S"]

    x = solve(M, data["z"])

    alpha = np.dot(data["z"],x)
    return alpha
```

(As you can see, I saved the given water matricies in a dictionary called `data`) Here is the table: (where are the chairs?)

ω	$\Delta\omega$	α
0.800	-0.0005	1.62781569
	+0.0005	1.64443120
1.146	-0.0005	994.752991
	+0.0005	-4185.01828
1.400	-0.0005	-2.71388011
	+0.0005	-2.69992680

Ut us clear that the solution is far more sensitive around 1.146.

As we are looking for the influence of a given perturbation of ω on the matrices, the (b) error bound is the the one we would like to use. The problem is, that this is not ascertaining anything regarding α. But if we are still allowed to assume z to be exact, we can simply find $\alpha = \mathbf{z}^T\mathbf{x}$ giving us:

$$\frac{\|\Delta\alpha\|_\infty}{\|\alpha\|_\infty} = \frac{\|\mathbf{z}^T\Delta x\|_\infty}{\|\mathbf{z}^T\hat{\mathbf{x}}\|_\infty} = \frac{\|\mathbf{z}\|_\infty\|\Delta x\|_\infty}{\|\mathbf{z}^T\hat{\mathbf{x}}\|_\infty} = \text{cond}(\mathbf{E}-\omega\mathbf{S})\frac{\|\delta\omega\mathbf{S}\|_\infty}{\|\mathbf{E}-\omega\mathbf{S}\|_\infty}\|\Delta x\|_\infty = \text{cond}(\mathbf{E}-\omega\mathbf{S})\frac{\|\Delta x\|_\infty\|\mathbf{S}\|_\infty}{\|\mathbf{E}-\omega\mathbf{S}\|_\infty}\|\delta\omega\| \tag{5}$$

Where we of course look at our own estimated/calculated $\hat{\mathbf{x}}$

e

1 & 2

As we are solving a matrix equation of the form $A\mathbf{x} = \mathbf{y}$. From what we have learned in linear algebra, this is only possible if A is non-singular.

A singular matrix has some signs. Its determinant is 0 (it is easy to see why this makes it impossble to solve the equation). As we might have noties from a(2), the condition number will also explode.

Looking at the plot below, it is clear that the system has a singularity around $\omega = 1.1463$, making $\alpha(\omega) \to \pm\infty$:

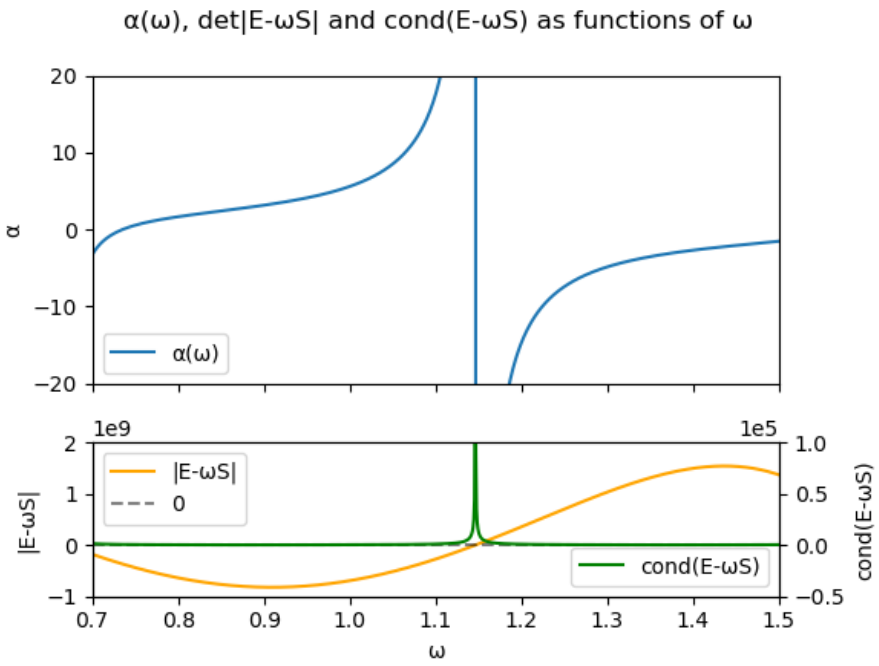

Figure 1: Eplot

f

1

```python
def householder_QR_slow(A):
    # some code has been removed here to not overwhelm the report
    for k in range(min(A.shape)):
        # Need to zero out the lower part of the k'th column to get in the right
    shape
        vk = np.zeros(m)

        # copy the lower part of the k'th column
        vk[k:] = R[k:, k].copy()

        # subtract "a_k" in the k'th column
        vk[k] -= -np.sign(vk[k]) * norm(vk)

        # we dont want to divide by zero
        beta = np.dot(vk, vk)
        if beta == 0:
            continue

        # make the transformation matrix
        H = I() - 2 * np.outer(vk, vk) / beta

        # apply it
        R = H @ R
        Q = H @ Q
    # As can be seen on page 124, we are actually constructing Q^T
    Q = Q.T
    # check that we did it right, as asked
    assert same(Q.T @ Q, np.eye(m)), "Q is not orthogonal"
    assert same(Q@R, A), "QR is wrong"

    return Q, R
```

2

To ease up on how much of my code I have pasted into this project, I will leave out the implementation of the faster version. The main ideas is the same, and the main headache came from placing V and R in the same array.

3

```
def solve_least_squares(A, b):
    # find V an R
    VR = householder_fast(A)
    R1 = np.triu(VR)[:A.shape[1],:]
    vs = np.tril(VR, -1)[1:,:]

    # we dont want to modify b
    c = b.copy()

    # We can simply use the vectors (no need for using a matrix)
    for v in vs.T:
        c -= 2 * v * np.dot(v,c)/np.dot(v,v)

    # solve R1x = c1 as described in the book
    c1 = c[:R1.shape[1]]
    x = backward_substitute(R1, c1)
    return x
```

Solving the "A1, b1"-example i get $\hat{x} = \begin{bmatrix} -0.000 \\ 0.500 \end{bmatrix}$ using the upper triangular matrix $\begin{bmatrix} -5.916 & -7.437 \\ 0.000 & 0.828 \end{bmatrix}$.
This is consistent with the "x1" that was supplied.

g

1

I (slightly arbitrarily) choose to look at ω less than 1.1 since the discontinuity is at 1.146-something

2 & 3 & 4

For N=4 i got the coefficients $[-29.20165559, 57.68267638, 60.35951443, -177.7027119, 94.21099494]$

For N=4 and N=6, below the fittet graph, the relative residuals and the calculated significant figures can be seen:

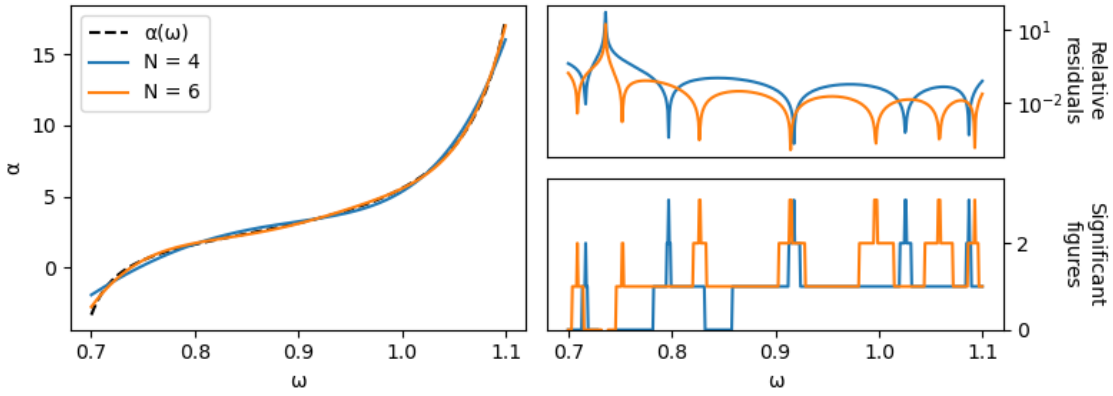

Figure 2: Caption

h

1

Q: Why can the g polynomial not approximate $\omega(\alpha)$?

A: Firstly, all the powers are even. Secondly, there is no way to 'simulate' the singularity (dividing by zero)

2

Taking **heavy** inspiration from the hint for my implementation I get the following results:

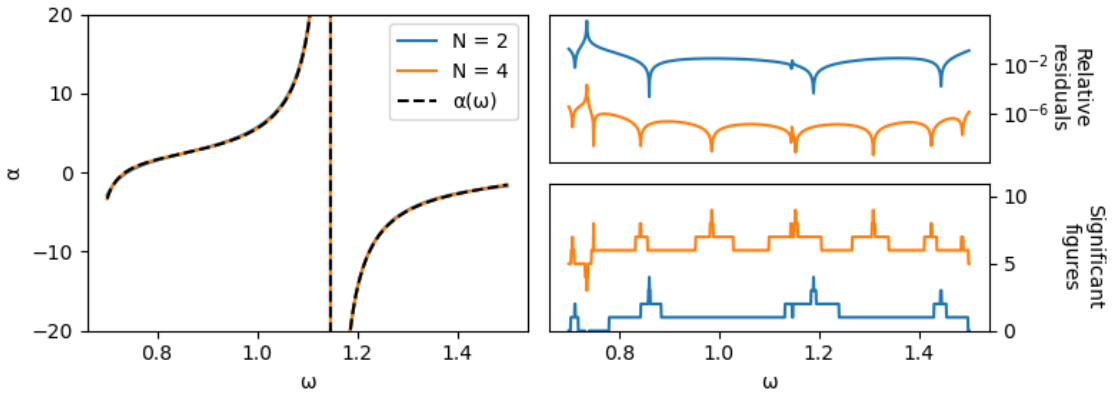

Figure 3: Caption

It is kind of hard to see the N=2 solution in the leftmost plot, but it is clear that having polynomials of fourth order greatly benefits the approximation.

3

In order to approximate the additional singularities, I could not come up with a smarter solution than simply allowing the polynomial in the denominator to have equally many roots (as can be seen when compared to only having a polynomial of fifth degree):

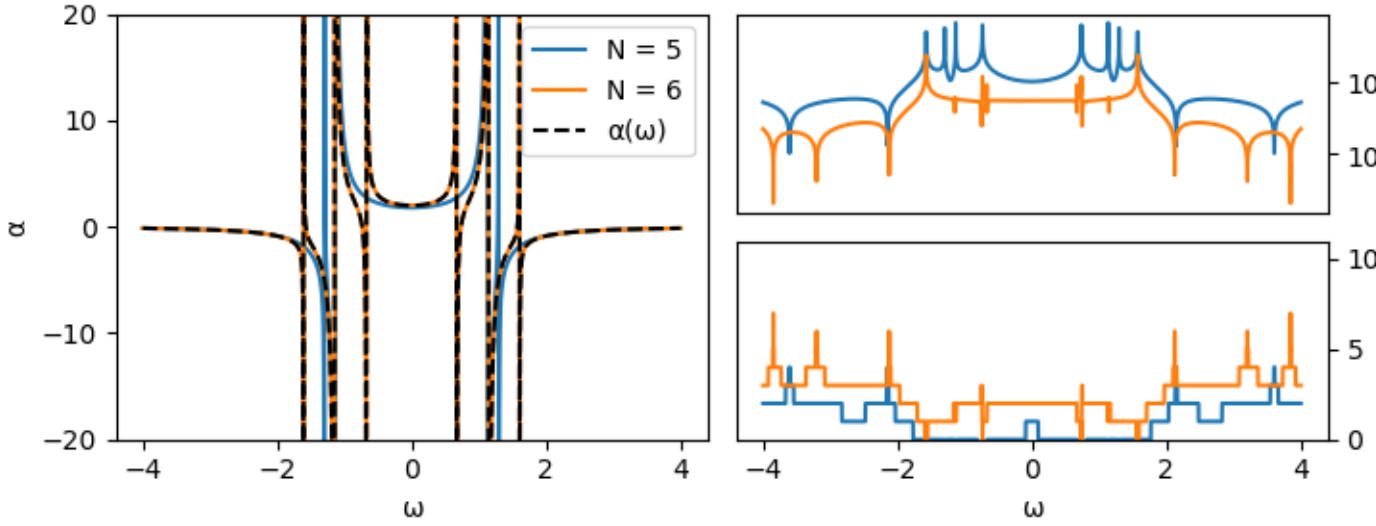

Figure 4: Caption